HOSPITAL COMPUTER PROJECT

A JOINT RESEARCH EFFORT OF THE MASSACHUSETTS GENERAL HOSPITAL AND BOLT BERANEK AND NEWMAN INC

MEMORANDUM NUMBER SIX-E

PROGRAMMING SOFTWARE STATUS REPORT

Submitted by Bolt Beranek and Newman Inc Cambridge, Massachusetts 1 May, 1966

NEK AND NEWMAN

· DEVELOPMENT · RESEA

HOSPITAL COMPUTER PRO

MEMORANDUM NUMBER S

ROGRAMMING SOFTWARE STATUS RE

1 MAY

EW YO CHICAGO LOS A

HOSPITAL COMPUTER PROJECT*

M E M O R A N D U M S I X E

Status Report on Programming Software

Submitted by
BOLT BERANEK AND NEWMAN INC
Medical Information Technology Department

Jordan J. Baruch, Principal Investigator, 1962-1966
Paul A. Castleman, Principal Investigator, 1966-
Richard H. Bolt, Supervisor, 1966-

Nancy Adams
John Barnaby
Sheldon Boilen
Richard A. Bolt
Michael Cappelletti
Jonathan Cole
Bernard Cosell
Edward Gilbert
Edith Grossman
Nancy Haggerty
Alice Hartley
Gray Hodges
Toby Jensen
Charles Jones
Larry Kershaw
Philip LaFollette
Nancy Lambert
Maureen Letu
Sanford Libman
Barbara Lieb
William Lucci
Richard Mahoney
Ernest McKinley
Charles Morgan
Andrew Munster
Nicholas Papias
Robert Paysor
Lawrence Polimeno
Sylvia Saraf:an
William Slack
Sandra Sommers
Lee Stein
Peter Storkerson
Sally Teitelbaum
Frederick Webb
Walter Weiner
Steven Weiss

* This joint research project is supported by Grant FR 00263 and Contract PH 43-62-850 from the National Institutes of Health and the American Hospital Association. Efforts contributed by the Laboratory of Computer Science, Massachusetts General Hospital, are directed by Principal Investigator G. Octo Barnett, M.D.
The report, Memorandum Six E, gives a technical description of programming software developed by BBN for the computer system being used on this project.

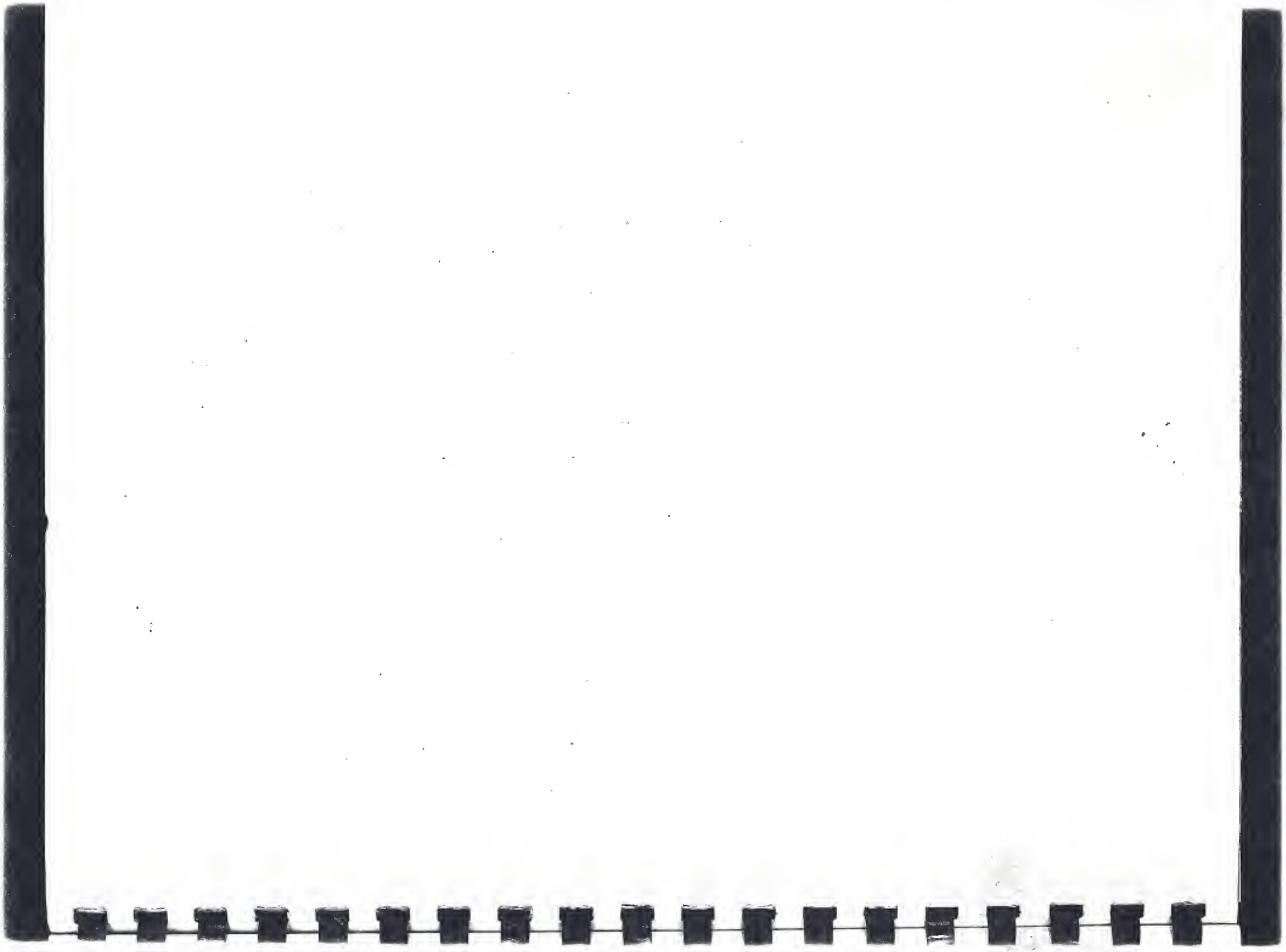
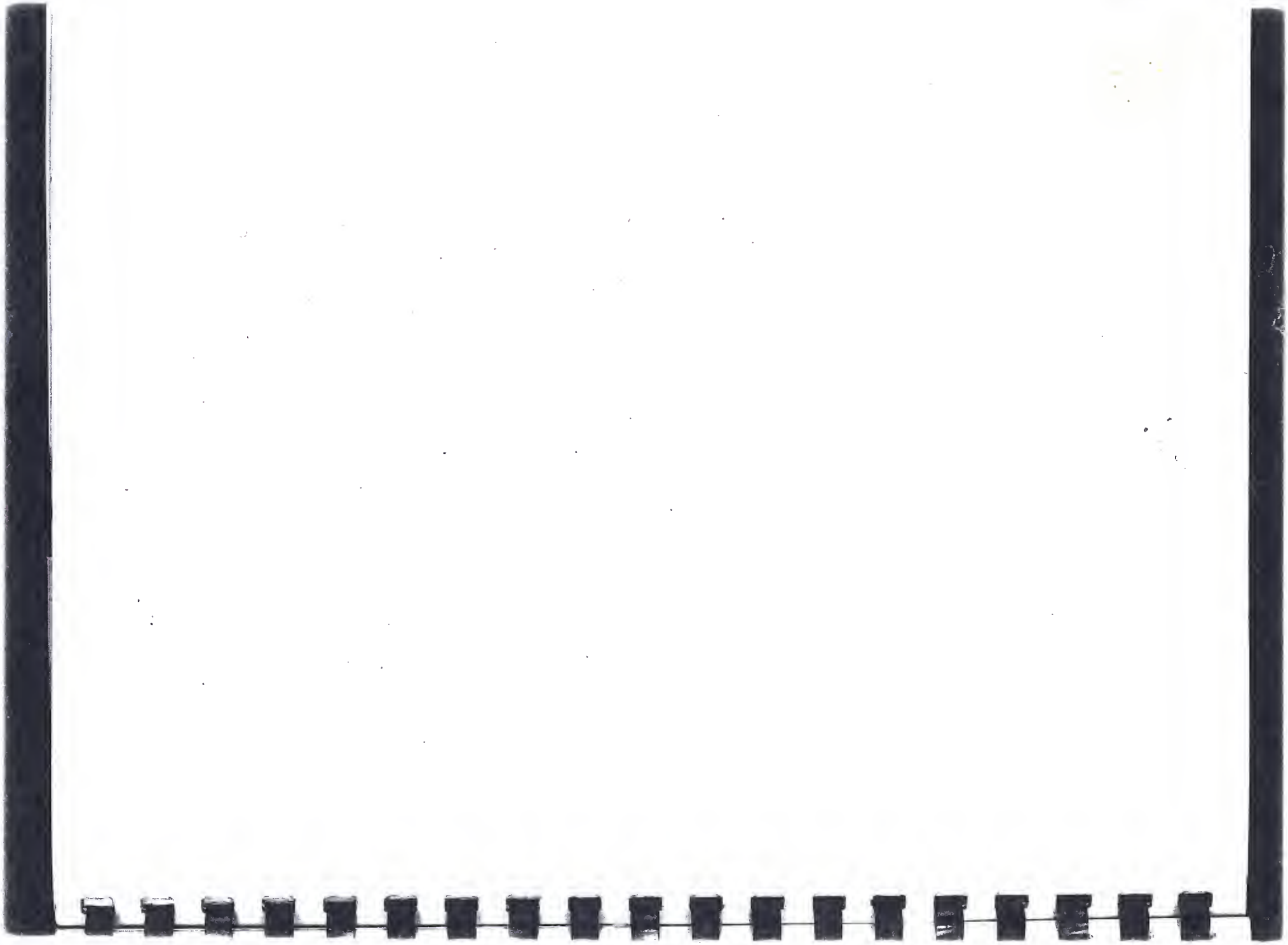


TABLE OF CONTENTS

TABLE OF CONTENTS cont.

TABLE OF CONTENTS cont.



HOSPITAL COMPUTER PROJECT
STATUS REPORT MEMORANDUM SIX E
PROGRAMMING SOFTWARE

INTRODUCTION

This is the fifth volume of Memorandum Six, a series of reports describing the Hospital Computer System developed by Bolt Beranek and Newman in collaboration with Massachusetts General Hospital, under sponsorship of the National Institutes of Health and the American Hospital Association. Other volumes in the series describe the system hardware, the time-sharing Executive and Common Routines, the hospital user programs, and the information storage and retrieval programs.

This volume describes the on-line Programming System that used to compose, edit, assemble, and debug software for the PDP-1b-45, the central processor designed for use in the collaborative project.

The current Programming System, with its full range of programming aids, is the realization of goals first approached at Bolt Beranek and Newman by the Simbug System. Simbug, one of the first operating time-shared systems in the general-purpose category, enabled several users to perform debugging operations simultaneously. The on-line Programming System described here accommodates a full staff of programmers and offers each of them, in effect, sole access to a general-purpose computer.

The system consists of four programs: Midas, a macro-assembler; Editor, a symbolic text editor; the DDT debugging program; and Handle, a program that provides for the manipulation of files generated by the three other programs in the system.

Chapter I contains a detailed description of the Midas assembly language. Midas offers the programmer a wide range of expression, permitting him to deal with individual computer words or to manipulate large blocks of computation.

Chapter II describes Editor, the on-line text editor used for preparing symbolic programs to be assembled by Midas.

Chapter III provides a description of DDT, the system's debugging and control program. DDT serves two purposes within the Programming System. First, it is used to check out and correct new programs that, for this purpose, run under its control. Second, DDT controls the running of Midas, Editor, and Handle, providing inter-program communication within the Programming System.

The final chapter, Chapter IV, describes Handle. The discussion of Handle includes, in addition to its actual functions and commands, a description of the file organization for the entire Programming System.

I. THE MIDAS ASSEMBLY SYSTEM

The Midas Assembly System described in this volume was adapted by Bolt Beranek and Newman from the Midas Assembler originally written for the PDP-1 at Massachusetts Institute of Technology. Some features of the Massachusetts Institute of Technology's system have been eliminated. Other features have been added to take advantage of the file structure of the Bolt Beranek and Newman system and to provide on-line assembly via Teletype terminal.

Like the Massachusetts Institutes of Technology's version of Midas, the Bolt Beranek and Newman system incorporates the basic features of the Macro Assembly Program. Macro was also developed at the Massachusetts Institute of Technology, first for the TX-Ø computer, and then for the PDP-1. The major departure that Midas makes from Macro lies in the handling of macro-instructions. A macro-definition in a Macro source program is partially assembled when first encountered; that is, everything except dummy symbols is translated into machine language and stored in the macro table. In Midas the actual source language text of the definition is stored in the macro table, and a complete assembly is performed each time the macro is used. This method significantly extends the system's macro instruction capabilities, permitting recursive and conditional definitions. The advantage gained by the Midas method for storing macro-definitions will become clear when macro-instructions are discussed in detail later in this section

The following pages assume that the reader has a working knowledge of basic PDP-1 mnemonic codes and their octal

representations. The reader who does not is referred to Volume Six A of this Memorandum, Chapter II, Section B, page 16.

A. INTRODUCTION TO ASSEMBLY

A problem that can be expressed quite briefly in words or mathematical notation will often, when put in the form of a computer program written in machine language, require hundreds of computer instructions. Since the value and location of each of these must be fully specified, the clerical effort involved in writing a long program may exceed that expended in analyzing the problem in terms of individual computer operations. Further inconveniences that arise from handling large amounts of information in the form of numerical code are the introduction of clerical errors into a correctly formulated program and the need for detailed documentation so that others may understand the coding.

Such difficulties led to the development of translating pro grams that make a computer available as an aid in the prepa - ration and documentation of the programs that will be run on it. These translators permit the programmer to express a computation in terms of a convenient mnemonic language (th source language), which the translating program is able to translate into machine code (the object language).

Two general categories of translating programs, compilers assemblers, have been developed to accept symbolic input a produce binary output. Compilers are translators that are "problem-oriented"; that is, the various compiling progra are designed to interpret a symbolic language similar to t language in which the problem would originally be stated. Thus, a compiler designed for mathematical applications w accept mathematical expressions, and in addition to trans ing the numbers and symbols into appropriate quantities, would translate the entire expression into an appropriate sequence of computer instructions. A PDP-1 compiler, for example, might translate the expression Z=X+Y into the se —LAC X, ADD Y, DAC Z. In the same way, compilers design for business applications are designed to accept a langua in which it is convenient to state business problems.

Because they accommodate various users in terms with whic they are familiar, compilers permit a person with a limit knowledge of computers to write his own programs. Assemb on the other hand, are "computer-oriented," consisting pr cipally of instructions which correspond to internal comp instructions. The simplest assemblers also include a mi number of rudimentary control operations that direct the translation process.

The Midas Assembly Program, while offering all the basic assembly features, belongs to a class of extended assembly programs referred to as macro-assemblers. Macro-assemblers such as Midas provide an extensive set of control operations (Midas pseudo-instructions) which, in principle, make it possible for the assembly program to perform computations analogous to those of any object program it can produce.

Most notable in this respect are the Midas macro-instruction features, which permit the programmer to define a special purpose abbreviative language to suit his own needs. Using pseudo-instructions provided for that purpose, a user can name a complex coding sequence and provide for varying parameters. The name is then, in effect, an abbreviation of the sequence that will be substituted for it at assembly time, with desired parameters inserted in their proper contexts. A programmer might, for example, take a sequence such as LAC X, ADD Y, DAC Z, assign it the name ADDXY, specify that X, Y, and Z are dummy symbols for which program symbols will be substituted as desired and then use merely the macro name ADDXY followed by appropriate symbols throughout the rest of the source program. The assembler will generate the predefined coding just as the compiler generates coding appropriate to the expression it is interpreting. Other pseudo-instructions available in Midas are ones that provide means for performing assembly-time list processing, symbol manipulation, and loop termination as well as for changing the course of assembly in response to certain conditions.

Formal constraints on the construction and manipulation of symbols are few, so the programmer may, within the range of processor capabilities, vary formats to suit particular programs. The programmer is free to ignore any of the spec features and use Midas as a simple mnemonic code translator

The formal rules of the Midas source language and basic prc cessor references are described in Section I-A.

Section I-B describes the functions and formats of all syst pseudo-instructions.

Section I-C explains the use of macro-instructions for performing an assembly.

Section I-D provides instructions for performing an assembl

Error conditions that are detected during an assembly and a sociated error messages are listed in Section I-E.

The notation used in this volume includes some special syml ↵ represents carriage return. →| represents a tab. Quotation marks indicate the pressing of the Control key o the Teletype keyboard. The symbols < > are used to enclos text that would normally be set off by quotation marks. U less otherwise noted, all integers appearing in the text a octal integers.

1. The Midas Source Language

A Midas source program is a string of alphanumeric and ope tional characters. From this string the Midas assembly pr gram produces the words that make up the object program an

places these words in their proper locations in memory. In order to accomplish this, the assembler must interpret the source program as a series of meaningful strings. In most cases, a string that is meaningful to the assembler represents a word in the object program. In other cases, the string may direct the assembler to produce several or no words in the object program.

This section describes the mechanics of creating legal Midas character strings, the references that the assembly program uses in associating character strings with binary values, the conventions that instruct Midas as to the type of value or actual value a string is to represent, and the overall source program format requirements.

The source program, described above as a single string of characters is, more precisely, a system of arbitrary strings, each of which consists of individual characters juxtaposed according to formal conventions. The construction of legal strings is hierarchical in nature. The lowest level constituent strings are formed from the alphabetic members of the character set. Higher level constituent strings are constructed from previously defined constituent strings using those members of the character set which function as combining operators. The type of object a constituent string represents is indicated by punctuation characters.

a. The Character Set

The complete character set from which the source language is constructed is included in Appendix A. It consists of all characters on the Teletype keyboard net specified as illegal.

The character set is generally divided into the following categories:

alphanumeric characters:	The letters A-Z and the character, <.>(period)*, which may be constituent of symbols; and the digi Ø-9, which may be consti ents of symbols or integ
combining operators:	Single characters repres ing fixed arithmetical logical operations to be performed by Midas.
punctuation characters:	These serve as string d limiters. A string- delimiting character ma serve a variety of purp depending on its use. String delimiters in ge eral identify individua strings and often indic the manner in which a s is to be interpreted. Appendix A for complete listing. Most generall delimiters are space, t and carriage return.

b. Legal Strings

(1) Basic Strings

The minimum character strings required to represent values the source program are symbols or integers, formed as foll

*The character <.> also denotes a decimal number (as in <1 and may also be used to represent the value of the current tion.

Integers

An integer is a string of digits (Ø,1,...9) that is evaluated as octal or decimal according to the radix prevailing at its appearance. The integer value is its representation as an 18.-bit binary number which restricts integer values to 777777 if the radix is set for octal and to 262143 for decimal. An integer above these limits will be evaluated modulo $(2^{18.}-1)$.

Symbols

A symbol is defined as a string of characters, the first six of which must distinguish it from all other symbols.

Letters, numbers, and periods may be used as symbol constituents, but at least one of the identifying six must be a letter.

Longer symbols, useful for mnemonic or documentary purposes, may be used, since the Midas processor ignores any character except a terminator in excess of six.

Symbols for macro names and pseudo-instructions are subject to the same restrictions as symbols for numerical values.

Note that the symbols <READIN> and <READINTAPE> are both legal symbols; if used in the same source program, they both will appear to the assembler as <READIN> and will be used interchangeably. If distinct symbols are desired, care must be taken to differentiate symbols within their first six characters.

(2) Complex Strings

(i) Language Units

Complex strings may be formed from basic strings by use of characters that are provided as combining operators. Altho integers and symbols are the only basic strings in the Mida language that are formed by purely alphanumeric concatenati a complex string may be bracketed and function in the same as a basic string in a new combination. In discussing the struction of complex strings, language units will be called either syllables or expressions as defined below.

Syllable

A syllable is any component string of an expression whose value is independent of its use in the expression. An expression enclosed in brackets may be used as a syllable to form other expressions. In addition, the character <.>, used to represent the current value of the location counter modulo (2^{12}.), functions as a syllable.

Expression

An expression is a string consisting of one or more syllables separated by combining operators.

(ii) Combining Operators

The characters listed below according to function, are the Midas combining operators. Quotation marks denote that th Control key must be pressed while typing the character.

Product Operators	Function
"T"	Folded integer multiplication
"X"	Logical disjunction (exclusive OR)
"U"	Logical union (inclusive OR)
"A"	Logical intersection (AND)
"Q"	Quotient
"R"	Remainder

Additive Operators	Function
+ or space	Addition, mod $2^{18.}-1$
-	Addition of the one's complement

Note that <A"T"B> results in an 18.-bit quantity equal to the sum of the unsigned magnitudes of the high and low order halves of the 36.-bit product produced by regular multiplication of the two 18.-bit quantities A and B. If both A and B are small enough, this function produces the ordinary product.

When evaluating expressions, Midas performs operations from left to right, all product operations preceding additive ones. If a string of consecutive additive operators occurs, Midas will perform only the last. A string of consecutive product operators, however, cannot be processed.

Examples. In the following examples, various equivalent expressions are shown, and their component syllables listed.

a)	Expressions:	LIO 1ØØ	1ØØ LIO	22Ø1ØØ
	Syllables:	<LIO>,<1ØØ>	<LIO>,<1ØØ>	<22Ø1ØØ>
b)*	Expressions:	LIO ADD	JSP	62ØØØØ
	Syllables:	<LIO>,<ADD>	<JSP>	<62ØØØØ>
c)	Expressions:	7-2"U"3	6-2	4
	Syllables:	<7>,<2>,<3>	<6>,<2>	<4>
d)	Expressions:	+A	A	2"T"A-A
	Syllables:	<A>	<A>	<2>,<A>
e)	Expressions:	A+B"T"C	A+[B"T"C]	
	Syllables:	<A>,<B>,<C>	<A>,<[B"T"C]>	

Note that the expression [A+B]"T"C is not equivalent to those of (e).

2. The Assembly Program

In order to interpret symbols and integers and to assign ther to memory locations, Midas must make references to the Curre Location Counter, the Symbol Table, and the Radix Indicator, which are described below.

a. The Current Location Counter

The Midas Assembler assign assembled words to sequential loc tions, starting from any given location. A register in the sembly program, referred to as the Current-Location Counter, indexed whenever a location is assigned, indicating the loca tion which will be assigned to the next word assembled. It

*Assume that LIO=22ØØØØ, ADD=4ØØØØØ, and JSP=62ØØØØ.

initially set at 11Ø and counts upward modulo (2^{16}.). Conventions are provided in the source language so that the programmer may assign a numerical value to the current-location counter, thus specifying the first location in any sequence. Using other conventions, the programmer may direct the assembler to increment the current location a specified number of times although no words are assembled.

A source language string may be specified as a direct representation of the value of the location counter at any time during the assembly and used in place of that value throughout the source program. In addition, a single character is provided that may be used to represent its value in any word while it is being assembled. Any value assumed by the current location counter is an actual location of a storage word. Representations of its value, however, are equivalent to the sum of the location plus a number referred to as the offset count. Unless reset by the programmer, the offset count will be Ø, and representations will coincide with the locations at which they are derived. The offset count may be set by the programmer, however, and an internally consistent set of values derived relative to the locations being assigned. The utility of offsetting the current location is discussed later in connection with the pseudo-instruction OFFSET. It is mentioned here to emphasize the fact that a number is assigned to a symbolic address tag in order to utilize a list position rather than to represent it.

While the fact that program addresses and computer instruction codes may be symbolically represented is sufficient knowledge for writing source programs, the fact that these represent numbers to the assembler, and locations and operations only

in context of the source program, will be useful to understanding the full range of symbolic manipulations that may be performed. For example, if a computation to be performed requires the number 1ØØ, we might <LAW> the number rather than provide a register containing it, and if the number 1ØØ is represented by the symbol <A>, we can <LAW A> whenever we need 1ØØ in the accumulator whether <A> derived its value as an address tag or as a parameter. As further illustration, one step in the computation might be to add the contents of register 1ØØØ to the value in the accumulator, requiring a word in the executable sequence containing the number 4Ø1ØØ. Given the symbol <ADD> representing the numerical operation code <4ØØØØØ>, the symbol <B>, which has been set to equal <1ØØØ> as a symbol address tag, and the operator space, which signifies addition, the source language expressions, <ADD B> or <B ADD>, would give the proper value. Storage of the expression in an executable sequence is accomplished by setting the expression in the proper context. If we need to perform a computation using the number <4Ø1ØØØ>, we might think in terms of creating a data word to contain it when we could just as well use the instruction word and would particularly wish to when computer space is limited. The source language takes advantage of the fact that a computer word may be an instruction, data, or both according to its context in the program in that strings, while associated with a number in one prescribed context, may then be generally applied.

While the advantages of contextual as well as explicit interpretation may be clear to programmers who are familiar with the techniques of machine language coding and who will be looking for these properties in the source language, others might well tend to think in terms of symbols for addresses and symbols for words as quite different entities.

b. The Symbol Table

Symbols, when associated with a value, are entered in the Midas Symbol Table, which is used as a reference by the assembler. Mnemonic symbols for PDP-1 instruction codes are part of the initial contents of the symbol table. (A list of these is included in Appendix B.) These and programmer constructed symbols, which stand for numerical values, are "substitutive" symbols. Two other symbol types are included in the symbol table: pseudo-instruction names and macro names. These are referred to as "operational" symbols; there is no single address or single 18.-bit number with which they are synonymous. All pseudo-instruction names are included initially in the symbol table, defined by a reference to an assembler routine. A macro name is entered in the symbol table when a macro-instruction is entered in the macro table, and a macro name is defined as a reference to the macro table location of the instruction to which it is assigned.

Midas assembles a source program in two passes; that is, two complete scannings of the source program are necessary to produce an object program. Midas was constructed this way so that symbols may be associated with values at any point in the source program without restricting their use prior to definition, thus permitting the programmer freedom from rigid format constraints.

c. The Radix Indicator

Integers are interpreted as octal or decimal according to the radix prevailing when they are encountered by Midas. A register in Midas, called the current radix indicator, is initially

set to 8 for accepting octal integers. Subsequent settings of 8 or 1Ø may be effected by source program specification.

3. Defining Symbols

The Midas symbol table, described earlier, functions as a d tionary during the translation process. Each symbol introd by the programmer is inserted together with its value into symbol table during Pass 1. The value of a symbol is refer to as its definition. Numerical definition of a symbol may accomplished by its appearance as an address tag, a variabl name, or in a parameter assignment. Symbols may also be de fined as macro-instruction names or in terms of other symbc Macro name symbols are discussed in the section on macro-instructions. The establishment of symbol synonyms is dis-cussed in Section II in connection with the pseudo-instruct EQUALS and OPSYN.

The three basic formats that Midas requires for assigning a symbol to a numerical value are described below.

a. Address Tags

A symbol is identified as an address tag if it is terminat by a comma or a colon. An address tag is equated with the value obtained by adding the value of the location counter when scanned to the offset count. The resulting value is entered into the symbol table modulo ($2^{12.}$) if terminated a comma, and modulo ($2^{16.}$) if by a colon. A symbolic addr tag identifies a line in the source program text and is re quired only for lines referenced within the text.

An expression, such as <A+3>, terminated by a comma or colon will be admitted for use as an address tag only if it coincides with the value of the current location counter plus the offset count. Since relative addressing using address arithmetic is an intrinsic system capability, an untagged word may always, in effect, be referenced by such an expression. If a previously defined symbol occurs as an address tag, the symbol will not be redefined and will be admissible only if its value agrees with that of the current location counter plus the offset count.

b. Variable Names

A programmer may direct the Midas assembler to reserve a sequence of storage words for variable quantities produced by a computation. Symbols may be substituted for these locations before they are known. The assembly program classifies such symbols as "undefined variables" and assigns them provisional relative values. A string is classified as an undefined variable by the inclusion of the character, #, in the identifying string itself, that is, within the first six characters, in at least one of its appearances. The symbol is stored in the symbol table without the qualifying character and may be referred to with or without it. Actual values are entered for all variables still undefined at the appearance of the pseudo-instruction VARIABLES. The following are legal variable names: <ABC#>,<#ABC>,<AB#C>. For regularity of documentation the second form listed, however, is usually used.

c. Assigned Parameters

A programmer may, with an identity declaration, assign a va to a symbol that is to be used as a parameter.

The format is

SYMBOL=EXPRESSION

The symbol to the left of the equals sign is entered in the symbol table, and the value of the expression to the right entered as its definition. If no value can be obtained fo: expression, the symbol is not defined.

4. The Use of Expressions

The rules for forming expressions were given earlier. The evaluation of an expression depends on its context in the source program as well as on the value of its component sy lables. Contexts in which Midas evaluates expressions are described below.

a. Storage Words

An expression terminated by a tab or carriage return is a storage word. A storage word, when encountered by Midas, is evaluated and assigned the memory location equal to the value of the current location counter. The contents of a storage word may ultimately be used as an instruction and, operated on by an instruction, depending on the use of the word in the object program.

b. Constants

Constant values required by a program need not be introduced as storage words in the source program. The constant expression desired, enclosed in parentheses, may appear literally as the operand of an instruction.

For example, an instruction to subtract 1ØØ. from the accumulator could be written as <SUB(1ØØ.)>. Midas will generate a word containing the value 1ØØ. The string <(1ØØ.)> is called a constant syllable, and the address of the word containing <1ØØ.> is substituted for it. Note that (1ØØ.), (5Ø.+5Ø.), or (A+25.), where A=75. are equivalent constant syllables and will all refer to the same location.

Constants may appear within constants to any depth, as in

(1) LAC(LAW(FLEXO ABC))

The right parenthesis of a constant syllable may be omitted if the constant is followed by a word terminator.

For example,

(2) LAC(LAW(FLEXO ABC)

is equivalent to (1) above.

Omission of the right parenthesis in

(3) LAC(LAW(ABC)-1)

would, however, change the meaning.

c. Location Assignments

A location assignment is an expression immediately followed a slash. When Midas encounters a location assignment, the e pression is evaluated, and the location counter is set to th value. If an expression that is used to assign a location c tains any undefined symbols when encountered by Midas on Pas. the current location becomes indefinite. This means that th definition of address tags is inhibited until a defined loca assignment occurs, and at that time the counter again become definite. On Pass 2, an undefined symbol in a location assi ment will cause an error message (USL). The undefined symbc is taken as zero, and the location remains definite. The Midas command, "E", discussed in Section I-D permits the prc grammer to arrange for Midas to type a message if the locati becomes indefinite on Pass 1.

5. Source Program Format

Midas begins processing a source program after it encounters title. A title may be any string of characters terminated b a carriage return. Initial carriage returns are ignored. T end of the source program is indicated by the appearance of the START pseudo-instruction.

The portion of the source program which is to be assembled, referred to as the body, is composed of character strings. These strings are processed by Midas sequentially.

Comments may be entered throughout the source program. Any string of text characterized as a comment will be ignored by the assembler. A comment is introduced by a slash and must be preceded and terminated by either a tab or carriage return.

Figure 1. Two sample Midas programs for solving the equation Y = 7A+B/2, for A = 4Ø, B = 6Ø. The symbolic programs in columns (1) and (2) will produce the machine language program in column (3).

(1)

```
SAMPLE PROGRAM 1

1ØØ/    LAC A
        MUL SEVEN
        RIR 1S
        DIO SEVENA
        LAC B
        SAR 1S
        ADD SEVENA
        DAC Y
        HLT
A,      4Ø
SEVEN,  7
B,      6Ø
SEVENA, Ø
Y,      Ø

START 1ØØ
```

(2)

```
SAMPLE PROGRAM 2

1ØØ/    LAC (4Ø
        MUL (7
        RIR 1S
        DIO SEVENA
        LAC (6Ø
        SAR 1S
        ADD #SEVENA
        DAC #Y
        HLT

CONSTANTS
VARIABLES
START 1ØØ
```

(3)

```
1ØØ   2ØØ11
1Ø1   54Ø1
1Ø2   672Ø
1Ø3   32Ø1
1Ø4   2ØØ1
1Ø5   675Ø
1Ø6   4ØØ1
1Ø7   24Ø1
11Ø   76Ø4
111   ØØØØ
112   ØØØØ
113   ØØØØ
114   ØØØØ
115   ØØØØ
```

PRINTOUT FROM ASSEMBLY:

(1)

```
DEFINED SYMBOLS ALPHABETIC

   A      111
   B      113
   Y      115
 SEVEN    112
SEVENA    114
```

(2)

```
CONSTANTS AREA,  INCLUS
      FROM         TO
      111          113

DEFINED SYMBOLS ALPHABE
      Y            115
   SEVENA          114
```

The strings <CONSTANTS>, <VARIABLES>, and <START> in the sa programs are pseudo-instructions, which will be discussed i the following section.

B. PSEUDO-INSTRUCTIONS

Pseudo-instructions are source-language expressions that serve to direct the assembly process. A pseudo-instruction statement consists of a pseudo-instruction symbol terminated by a delimiter and followed by arguments as required. Unless otherwise noted, a pseudo-instruction statement is terminated by a tab or a carriage return. Pseudo-instruction symbols, like all other symbols, are identified by no more than six characters. Thus, pseudo-instruction symbols composed of more than six characters may always be abbreviated. For example, DIMENSION may be shortened in use to DIMENS. The pseudo-instruction repertoire is described below with regard to format and function.

1. OCTAL and DECIMAL

When integers are encountered, they are interpreted as octal or decimal according to the value of the prevailing radix indicator. The pseudo-instructions OCTAL and DECIMAL reset the radix indicator, which is set by Midas to OCTAL at the beginning of each pass. An integer syllable followed directly by a period will be interpreted as a decimal number regardless of the current radix and will not change the radix value.

2. CHARACTER, FLEXO, and TEXT

These three pseudo-instructions were originally devised for storing 6-bit Friden code characters. The translation of 7-bit Teletype code to Internal Code may result in 6 or 12. bits. It has been left to the programmer to introduce 12.-bit code only where two 6-bit characters are permitted. A table of characters and their Internal Code is included as Appendix C.

The pseudo-instruction CHARACTER is used to place a characte code in the left, middle, or right 6-bit portion of an 18.-b word. The symbol, CHARACTER, is followed by L, M, or R acco ing to desired bit position, immediately followed by the cha acter to be coded. Characters with 12.-bit codes cannot be stored this way.

The format is	Stored as:
CHARACTER RA	ØØØØ41
CHARACTER MB	ØØ42ØØ
CHARACTER LC	43ØØØØ

The character strings shown above are pseudo-instruction sy lables and may be used in the same manner as symbols or int gers. For example,

LAC (CHARACTER LA + DTB) is equivalent to
LAC (41ØØØØ + DTB.

The pseudo-instruction FLEXO is used to specify an entire computer word filled with character code, either three 6-bi characters or one 12.-bit and one 6-bit.

The format is	Stored as:
FLEXO ABC	414243
or	
FLEXO A←	417776

The pseudo-instruction TEXT is used to assemble a string o characters by groups of two or three, as appropriate, into successive words in the object program.

A TEXT statement consists of the symbol <TEXT> terminated by a delimiter and followed by a string of characters. The first character in the string is used as a delimiter of the text string itself; it is not stored as part of the text string, and its reappearance terminates code storage. Thus, if given the string <TEXT /MESSAGE/>, Midas will store character code for the word, MESSAGE. A 12.-bit code character is not a good choice for a text delimiter. Only six-bits at either end of a string are interpreted as delimiters, so the remaining six bits of a 12.-bit character would be included in the stored text.

If the character <#> is used in the argument of CHARACTER, FLEXO, or TEXT, it is handled in an exceptional way; it is stored as end-of-message (Internal Code 74) rather than as its own Internal Code configuration, Ø3. Consequently, there is no provision for entering <#> as actual text. Most hospital user programs use the code, 74, as a text terminator; it may appear in no other context.

Three pseudo-instructions--CONSTANTS, VARIABLES, and DIMENSION--are provided to direct automatic storage assignment of words for constant and variable data. Variable data words may be generated individually by reference or as fixed length arrays obtained with the DIMENSION pseudo-instruction. Constant data words are generated to accommodate literal references.

3. CONSTANTS

The pseudo-instruction CONSTANTS effects the allocation of constant syllables to storage words containing constant values,

beginning at a location whose value is equal to that of the current location counter at the appearance of CONSTANTS. W a constant syllable is allocated, its address is substitute for it at all references. If different expressions enclose within parentheses have the same value, they are considered be the same constant syllable and are associated with only location. The pseudo-instruction CONSTANTS may be used no than ten times in the same program.

Since storage space for constants is allocated on Pass 1 wh some expressions may not be definite, the number of registe reserved in the constants area may exceed the number Midas needs when all references have been consolidated; thus, a g of unused registers may arise between a constants area and subsequent portion of the object program.

The following examples show symbol prints following Pass 1 and Pass 2 for the same program.

CONSTANT AREA RESERVED, INCLUSIVE

FROM	TO
325	337

indicates registers reserved during Pass 1.
At the completion of Pass 2, the printout

CONSTANTS AREA, INCLUSIVE

FROM	TO
325	332

indicates those registers actually containing constants.

4. VARIABLES

The value of the current location counter at the appearance of the pseudo-instruction VARIABLES on Pass 1 marks the beginning of the storage area allocated to variables. All variable names still classified as undefined are assigned locations at this time. The relative value assigned to an undefined variable is added to the value of the first location in the sequence and the result obtained entered as its absolute address. Each defined variable represents an address at which Midas assigns a storage word whose contents are unspecified.

When a variables area has been completely allocated, the value of the current-location counter is that of the next location at which a storage word will be assembled. If VARIABLES appears when the location counter is indefinite, it is inadmissible. Thus it is wise for a programmer to use Midas command "E" (described in Section I-D) to arrange a printout if the location becomes indefinite on Pass 1 so that he can correct the condition before the processing of VARIABLES.

In the current version of Midas the pseudo-instruction VARIABLES may be used no more than ten times. If the maximum is exceeded, the error comment TMV (too many variables) is typed. The number of defined variables, however, is limited only by the capacity of the symbol table.

At the occurrence of VARIABLES on Pass 2, Midas compares the value of the current location counter with the value which was associated with that variables area on Pass 1. A disagreement is noted by the error message VLD (variables location disagrees), which indicates that subsequent symbol definitions or macro-expansions have altered the sequence of assembled words.

5. DIMENSION

The pseudo-instruction DIMENSION reserves in the variables storage area blocks of registers, which may be referenced relative to a single symbolic address. DIMENSION is used tc set up fixed-length arrays. The pseudo-instruction symbol a the name and extent of any number of arrays constitute a DIMENSION statement according to the following format:

DIMENSION NAME1(LGTH1),NAME2(LGTH2)

where the entire statement is terminated by a tab or carriag return and requested blocks are separated from one another l commas. The array name must be a legal symbol that has not been previously defined. The extent must be stated as an e: pression whose syllable values are known when the DIMENSION statement is encountered on Pass 1.

6. EQUALS and OPSYN

The pseudo-instructions EQUALS and OPSYN permit a user to e lish symbol synonyms, representing the same value. The for is

EQUALS SYNONYM,SYMBOL
or
OPSYN SYNONYM,SYMBOL

where <SYNONYM> must be a legal symbol string and the <SYME with which it is identified must be previously defined. <SYNONYM> is assigned the same numerical or operational val

as <SYMBOL>, and the two are thereafter synonymous. OPSYN effects this on Pass 1 only; EQUALS, on both passes. The following example illustrates the difference between the two.

Let us say, for example, that a programmer wanted to use the macro-instruction facilities to redefine a pseudo-instruction such that the new instruction was a function of the old. The original pseudo-instruction would have to be represented by a different symbol; otherwise, its appearance in the definition would act as a macro call, resulting in a closed loop.

For example, if one writes

EQUALS CHAR, CHARACTER

and then defines a macro-instruction, CHARACTER, in terms of CHAR, which now calls the pseudo-instruction, on Pass 2 CHAR will again be made equivalent to CHARACTER, which has been redefined. CHAR then no longer references the pseudo-instruction, and the loop avoided on Pass 1 will occur anyway on Pass 2. If one uses OPSYN, however, CHAR will be associated with CHARACTER as desired on Pass 1 only and retain its identity with the original pseudo-instruction on Pass 2.

7. NULL

The NULL pseudo-instruction performs no action, but it is used as a substitute for symbols no longer needed in a program.

Some programs are required to be compatible with various environments (different machines, data bases, etc.), and a function performed frequently in one usage may not be performed at

all in another. For example, a symbolic program may contain complex macro-definitions that need not be assembled in one instance but must be available for other processings. In this case, the macro names may simply be equated with NULL, as in

```
EQUALS MACRO,NULL
```

Another case in which NULL is useful arises in connection with macro-table storage space and the "garbage collector."* When the amount of space available for the storage of macro-definitions is exhausted, the garbage collector will search the table for definitions which no longer have a reference in the symbol table and will recover such space by consolidating the remaining table entries.

The symbol-table reference to a macro that has been redefined is automatically transferred to the latest definition. In the case of macros that have not been redefined but are simply no longer needed, the reference must be suppressed in order to notify the garbage collector of the available space.

8. OFFSET

The pseudo-instruction OFFSET is used to set the value of the offset count, whose relation to the current location counter was described in connection with symbol definition.

---

*The garbage collector will soon be available for this system Available macro storage space is $61,44\emptyset_{10}$ computer words (approximately $18\emptyset,\emptyset\emptyset\emptyset$ characters).

The psuedo-instruction format is

OFFSET EXPRESSION

where the value of the expression (positive or negative) is stored as the offset count. When the offset count is any value other than zero, a symbol value derived as an address tag will not equal the core location of the storage word with which it is associated. For example, the coding:

```
OFFSET 5
ABC,    LAW 1ØØ
        JMP ABC
```

occurring when the current location counter contains 1ØØ will be assembled as:

```
1ØØ,  LAW 1ØØ
      JMP 1Ø5
```

A portion of the object program that was assembled under these conditions is not executable at the location it occupies if storage word expressions use these symbols as referents. The offset capability is, however, useful in creating a body of data independent of its core location, yet internally consistent.

The effect of one OFFSET declaration is terminated by the appearance of another. If a return to the normal sequence is desired, the programmer must set the offset count to zero. OFFSET is used in connection with memory "renaming" and in constructing item maps.

9. REPEAT

REPEAT instructs Midas to assemble a specified portion of t source program a specified number of times and thus relieve the programmer of the necessity for source language repetit of a repetitive object program sequence.

The format is

REPEAT EXPRESSION,TEXT

where EXPRESSION is the count of the REPEAT, specifying the number if iterations desired, and the TEXT is the source pr gram section to be iterated, called the range of the REPEA The count must be defined when Midas encounters the REPEAT Pass 1; otherwise, the range is ignored and the error prin <USR> occurs.

A carriage return is used to terminate the entire instruct tabs are used within the range to denote storage words. T may also appear within a macro-definition or as an argumen Brackets may be used to enclose portions of the range or t entire range so that carriage returns may be included.

Since a REPEAT merely serves to reproduce a string, the ra may include any elements of the source language, including other REPEATS and macro calls. An internal REPEAT, unless is at the end of the range, must be bracketed; otherwise i terminating carriage return would also terminate the first REPEAT.

In the following example, the statement

REPEAT 2, LAC A →| ADD B →| DAC C

will generate for assembly the coding

```
LAC A
ADD B
DAC C
LAC A
ADD B
DAC C
```

If the count of a REPEAT is zero or negative the range is not processed.

10. START

The START pseudo-instruction directs Midas to stop reading characters. START must appear at the end of every source-language file and may take as an argument an expression denoting the starting address of the object program.

The format is

START EXPRESSION↵

At the end of Pass 2 in response to command "J", (Section I-D), Midas appends to the binary output a word containing <JMP EXPRESSION>, where EXPRESSION is the argument of the last START processed.

11. EXPUNGE

The pseudo-instruction EXPUNGE removes symbols from the symbo table.

The format is

EXPUNGE SYM1,SYM2,SYMN

where the argument is a list of symbols, separated by commas and terminated by a tab or carriage return. Any type of symb may be expunged. Midas ignores undefined symbols in the list If any member of the list is not a legal symbol, Midas ignore the rest of the list. An expunged variable will not be defin unless it appears again with <#> after the EXPUNGE; <#> itsel may not appear in the argument list.

12. WORD

The pseudo-instruction WORD appends 18.-bit computer words, specified by the argument(s) of the pseudo-instruction, to a block of binary output.

The format is

WORD EXPRESSION
or
WORD EXPR1,EXPR2,...EXPRN

The appended words are not necessarily part of the object program; their values are selected to produce special binary formats when needed. For example, words might be appended i

order to accommodate a particular loader or to insert jump blocks before the end of assembly. Normal binary output format is discussed in Section I-D.

13. ØIF and 1IF

A programmer may find it useful, particularly when handling comples macro-instructions, to be able to test the value of an expression and to condition part of the assembly on the result. Such testing is effected by the pseudo-instructions ØIF and 1IF, in conjunction with symbols called qualifiers, which represent tests available. The tests are as follows:

Qualifier	Condition is true if:
VP	the evaluated expression is greater than or equal to ±Ø
VZ	the evaluated expression is equal to ±Ø
P	Pass 2 is being performed
D	the expression tested is a defined symbol
N	the argument contains no characters (usually a dummy symbol of a macro or IRP)

The format of conditional statements is

ØIF	VP	EXPRESSION
1IF	VZ	EXPRESSION
ØIF	D	SYMBOL
1IF	N	SYMBOL

where the test requires an argument, and otherwise, ØIF P

The value of 1IF is 1 if the condition is true, Ø if false; the value of ØIF is Ø if the condition is true, 1 if false.

A conditional statement may be terminated by tab, carriage return or comma. A conditional value may be used as a syllable; in this case the conditional must be terminated by a slash.

For example:

LAC (ØIF VP X/+3

is equivalent to

LAC (3) or LAC (4)

while

LAC (ØIF VP X+3

is equivalent to

LAC (Ø) or LAC (1)

ØIF and 1IF are often used to obtain a zero or one as the count of a REPEAT.

For example:

(1) END: Ø

REPEAT ØIF VP 7777-END, PRINTX /OVERFLOWED CORE/

The address assigned to END (colon indicating address modulo 2^{16}.) is subtracted from 7777. If 7777 is greater, the test is true, and the value of ØIF will be Ø; thus, the count of the REPEAT will be Ø, and the message will not be printed.

(2) REPEAT 1IF P, EXPUNGE TYO,TYI,ONE

Example (2) will on Pass 2 direct Midas to expunge the listed symbols.

14. PRINT and PRINTX

The pseudo-instructions PRINT and PRINTX effect an on-line printout by Midas during assembly. These instructions are particularly useful for obtaining information during the processing of complex macro-instructions.

The format is

PRINT or PRINTX TEXT

where the argument may be text of the form used with the pseudo-instruction TEXT or, if used in a macro-instruction, dummy symbols.

PRINT will cause Midas to type a line in the same format as first three columns of an error listing (described in the se on error checking). The code PNT is substituted for an erro code in the first column and is followed by the argument and a terminal line feed.

PRINTX causes Midas to type only the argument. Since both pseudo-instructions are effective on both passes, a repetiti printout can be avoided only if conditioned, using ØIF or 1I with the qualifier P. For example, in response to

REPEAT ØIF P, PRINT /TEXT/,

Midas will print only on Pass 1.

15. STOP

The pseudo-instruction STOP is used when the programmer wis to arrest the expansion of a macro-instruction, an IRP, or range of a REPEAT. In any other context, STOP is ignored b Midas.

Within the range of a REPEAT, STOP will halt the expansion all subsequent text unless the REPEAT occurs within the bod of a macro-instruction or an IRP. In that case STOP, wheth in a REPEAT range or not, will suppress subsequent coding until the occurrence of the next TERMINATE or ENDIRP.

STOP may be used conditionally as in the following:

```
REPEAT 3, [REPEAT 1IF VZ A-B,STOP
A=A-B]
```

The text A=A-B will appear for processing up to three times. However, if A=2 and B=1 at the start, the count of the inner REPEAT, which generates the STOP, will have the value one before the second appearance, and the expansion of the first REPEAT will be arrested. STOP may also be supplied as an argument for an IRP or a macro call.

The remaining pseudo-instructions--DEFINE, TERMINATE, IRP, IRPC and ENDIRP--are described in the section on macro-instructions.

C. MACRO-INSTRUCTIONS

A macro-instruction is any legitimate source-language text that a programmer names and sets up so that when the name appears in the subsequent source program, Midas will assemb the text. The text and macro name are established by a mac definition, whose format is described below. Where the tex includes parameters that may differ with each occurrence of macro-instruction, these parameters may be represented by c symbols.

1. Macro-Definitions

A macro-definition is initiated by the pseudo-instruction DEFINE, delimited by any terminator. DEFINE is followed by macro name. A macro name must be a legal symbol, which, i previously defined, will be redefined. The macro name is f lowed by a list of dummy symbols, if needed, and terminate a tab or carriage return. If dummy symbols that appear in text of the macro-instruction are not listed after the mac name, Midas will treat them as ordinary symbols. After a macro name Midas interprets the first character other than space as the first member of the argument list. The argum list is discussed in greater detail later. Midas consider text following the name and argument line to be the body o macro-instruction. Midas stores this text until the appea ance of the pseudo-instruction TERMINATE, which signals th end of the definition. The body of the macro may include element of the source language, including other macro-definitions or calls. Any dummy symbol from the list may appear as a syllable in the body of a macro-definition.

a. Basic Format

The basic format of a macro-definition is illustrated by the following examples.

(1)
```
DEFINE
        ABSOLUTE    -   (MACRO NAME)
        SPA         }   (BODY OF THE MACRO)
        CMA         }
    TERMINATE
```

The macro name, <ABSOLUTE>, subsequently serves as a macro call in the source program. Midas will assemble the body of the macro (<SPA> and <CMA>) into the object program at every appearance of the macro call.

(2)
```
DEFINE
        SUM A,B,C
        LAC A
        ADD B
        DAC #C
    TERMINATE
```

(The character # must be the first character if it is used in a dummy symbol string.)

The macro call <SUM XORG,XINC,XMAX> will cause the following sequence to be assembled.

```
LAC XORG
ADD XINC
DAC #XMAX
```

b. Dummy Arguments

A programmer may use as many distinct symbols as desired as dummy arguments in a macro-definition as long as each appears in the dummy argument list. Members of the argument list are usually separated from one another by commas.* The position of an argument in the list is the model for the order of arguments supplied at a macro call.

Some syllables, although they are referenced only within the body of the macro, will represent a different value at each call. Such syllables may be represented by dummy symbols as specified in the argument list as generated arguments, for which Midas will automatically provide a symbol. A list of those dummy arguments for which Midas must generate symbols is preceded by a slash and follows the list of arguments for which the programmer must supply symbols as shown below.

```
DEFINE MACROSYM A,B/C,D,E
or
DEFINE MACROSYM /A,B,C
```

where all symbols are to be generated.

Symbols generated and inserted by Midas are of the form <..A∅1>, <...A∅2>, <...A∅3>, etc. If at a macro call the programmer supplies a real argument in a list position corresponding to that of a generated symbol, Midas will accept the supplied

*The following delimiters are also acceptable: +, -, space, ∪, ∩, ~, tab, =, (,).

symbol rather than generate one. A generated symbol may be used to define a variable in the text; Midas will generate a symbol of the form <#...AØ1>.

The following examples give an idea of the use of generated arguments.

1) DEFINITION

```
DEFINE CLEAR A,N/B
       LAW A
       DAP B
B,     DZM
       IDX B
       SAS (DZM A+N
       JMP B
TERMINATE
```

CALL: CLEAR TAB,1ØØ

EXPANSION

```
       LAW TAB
       DAP ...AØ1
...AØ1, DZM
       IDX ...AØ1
       SAS (DZM TAB+1ØØ
       JMP ...AØ1
```

2) DEFINITION

```
DEFINE SAVEAC /A
       DAC #A
       JSP SUBR
       LAC A
TERMINATE
```

CALL: SAVEAC

EXPANSION

```
       DAC #...AØ1
       JSP SUBR
       LAC ...AØ1
```

If an argument is supplied at the call, as in <SAVEAC TEMP>, the expansion will be

```
DAC #TEMP
JSP SUBR
LAC TEMP
```

Midas, when scanning the body of a macro-definition for dumm arguments, compares each legal symbol in the text with the symbols in the dummy argument list. Those symbols that correspond to any dummy symbol are stored in a special way, as described in Section I-C-3, Storage of Macro-Instructions.

If the programmer wishes to represent only a part of a symb by a dummy arguemnt, he may use an apostrophe to denote thi in the body of the macro definition.

In the pseudo-instruction <CHARACTER RA> the string <RA> sa fies the requirements for a legal symbol. Midas, however, stands its special meaning within the context of the pseudo instruction. During the dummy symbol scan, however, Midas would interpret RA as a single symbol unless an apostrophe used to indicate that <A> alone is a dummy symbol, as in

```
DEFINE MACRO A
    LAC (CHARACTER R'A
```

The apostrophe is deleted when the macro-instruction is defined. In the case of a nested macro-definition, apostrop are also deleted at the time of definition; that is, when higher level macro is called.

2. Macro Calls

A macro call consists of a macro name followed by a list o arguments separated by commas. The call is terminated by tab or carriage return.

The arguments of a macro call may include any character string (including an empty string) with the following restrictions. Since comma terminates an argument and tab or carriage return terminates the list, these may be included only in arguments enclosed by brackets. Brackets must be used in pairs and may be used within other brackets. Midas will consider all but the outermost pair to be part of the argument.

At the appearance of the macro call, Midas processes the body of the macro (stored in the macro table) as though it had appeared in sequence. At this time Midas substitutes for the corresponding dummy arguments and creates the correct number of generated arguments.

If the programmer supplies extra arguments at a macro call and the definition specified generated arguments, the extra supplied arguments will take the place of generated ones. If, however, arguments are supplied in excess of the total number (supplied and generated), the excess arugments are ignored. Note that Midas will not generate a symbol when a programmer fails to supply one that has been specified in the definition.

3. Storage of Macro-Instructions

After the occurrence of the DEFINE pseudo-instruction, Midas saves the name of the following macro-definition and scans the list of dummy arguments, keeping count both of the total number of arguments and the number of these arguments that are to be generated. While stored in the body of the macro in the macro table, Midas scans the text for dummy symbols. When Midas encounters a symbol that matches a symbol from the dummy argument list, the list position of the corresponding dummy argument is stored in place of the symbol in the text and is distinguished by a code prefix.

Macro-definitions within the body of the macro are stored literally and defined only when the instruction containing t is called.

When Midas encounters the final TERMINATE, the number of wor that were required to store the definition is deposited in t first macro-table register preceding the text. The macro na and the table location of the definition are entered in the symbol table.

4. Nested Macros

It is convenient when discussing nested macro-instructions t think of DEFINES and TERMINATES as if they were parentheses, the outermost pair constituting the highest level macro-definition. When the programmer calls the highest level ma definition, Midas stores the second level definition in the macro table, and so on. Internal macro-definitions may con dummy arguments of higher level ones. These arguments will be replaced by supplied arguments when the higher level def tion is called. Pairs of DEFINES and TERMINATES must count out. To ensure that they do, the programmer may use macro name as the argument of a TERMINATE instruction. Then if t DEFINE associated with that TERMINATE refers to another mac name, the error print <MND> (macro name disagrees) will inf the user of a "mispairing" of DEFINES and TERMINATES.

A series of examples of nested macros follows. Note in ex-ample M-I the use of apostrophe and the insertion of a supp argument into a nested definition.

M-I

```
DEFINE FLOAT INSTR
  OLD'INSTR-INSTR
DEFINE INSTR X
  LAW X
  JDA F'INSTR
TERMINATE INSTR
TERMINATE FLOAT
```

For example, if FLOAT MUL appears, the expansion will be

```
OLDMUL-MUL
DEFINE MUL X
  LAW X
  JDA FMUL
TERMINATE MUL
```

This macro-instruction may be used to change PDP-1 instructions to subroutine calls. Their original meanings could be restored by

```
DEFINE UNFLOAT INSTR
  INSTR=OLD'INSTR
TERMINATE
```

M-2

```
DEFINE MACRO X,Y
  LAC X
DEFINE MAC2 Y
  ADD Y
TERMIN
TERMIN
```

The call <MACRO ONE,TWO> will generate

```
LAC ONE
DEFINE MAC2 TWO
  ADD TWO
TERMIN
```

The argument supplied for <Y> at the call of MACRO must be a symbol, since it will be inserted as a dummy argument in the definition of MAC2.

M-3 It is usually safer to use rather meaningless symbols a dummy arguments to avoid duplication of real arguments.

For example,

```
DEFINE MACRO X
  LAC X
DEFINE MAC2 COUNT
  ADD (X+3
  DAC COUNT
TERMIN
TERMIN
```

If COUNT is also a program symbol that the programmer inadvertently supplies at the call of MACRO, the result would b

```
LAC COUNT
DEFINE MAC2 COUNT
  ADD (COUNT+3
  DAC COUNT
TERMIN
```

Example M-4 below illustrates a macro-instruction that redefines itself when first called.

M-4

```
DEFINE INCREM
  DZM X
DEFINE INCREM
  LAW 1Ø
  ADD X
  DAC X
TERMIN
TERMIN
```

At the first call of INCREM the following text is generated:

```
DZM X
DEFINE INCREM
  LAW 1Ø
  ADD X
  DAC X
TERMIN
```

Subsequent calls will generate

```
LAW 1Ø
  ADD X
  DAC X
```

5. The Pseudo-Instructions IRP and IRPC

The pseudo-instruction IRP (indefinite repeat) generates sequential iterations of text a number of times determined b the analysis of its arguments. A different set of argument substituted at each iteration.

An IRP statement consists of the <IRP> symbol followed by a list of arguments, each enclosed in brackets, terminated by tab or carriage return. Following the argument list is the body of the IRP that, like the body of a macro-definition, include any source language elements, including other IRP's and macro calls or definitions. The body of an IRP is delimited by the pseudo-instruction ENDIRP.

Each argument of the IRP is itself a list of subarguments separated by commas or carriage returns. The first two mem bers of a subargument list are dummy arguments, and each ma represent a syllable in the body of the IRP. The remaining members of the list are the real arguments of the IRP. Upo encountering an IRP, Midas processes the body of the IRP re peatedly, with different symbolic equivalents substituted f the dummy arguments each time according to the following pr cedure. Midas begins by substituting the first member of t real argument list for the first dummy symbol and the remai of the real argument list for the second dummy symbol. The remainder of the real argument list is then treated as the real argument list in subsequent processings until all lis are exhausted.

IPRC operates exactly as does IRP but on a different type list. Elements of an IRP subargument list are separated b

commas and may include a text string or a bracketed expression. Real arguments of an IRPC subargument list are not separated by commas; each character in the string is treated as an individual member of the list.

The following examples illustrate the use of IRP and IRPC. Notice that the second dummy symbol may be omitted if not referenced in the body, although its position must be retained by a comma.

```
I-1.  IRP [NAME,,RPA,PPA,TYO,TYI],[VALUE,,1,5,3,4]
        NAME=IOT VALUE
      ENDIRP
```

This IRP will effect symbol table entries for the listed instruction symbols and their corresponding IOT commands.

```
I-2.  DEFINE TYPE DIGIT
        IRPC [NUM,,Ø123...9]

        REPEAT 1IF VZ DIGIT-NUM,PRINTX /NUM/
        ENDIRP
      TERMINATE
```

Example I-2 shows use of an IRPC within a macro-definition. The digit supplied at the macro call will, during the expansion process, be compared with each digit in the subargument list until its equal is found and printed.

Example I-3 illustrates how to use the second dummy symbol, which represents a list.

```
I-3  DEFINE MACRO LIST
       IRP [X,Y,LIST]
       REPEAT ØIF  D X,MAC2[Y]
       ENDIRP
     TERMINATE
```

The list obtained for Y in the first IRP repetition is used a supplied list for another macro-instruction.

Example I-4 shows a series of nested IRP's used to define a macro-instruction that, given the list <X1,X2,...XN>, will up a matrix of the form:

```
X1,X2,......XN
X2,X3,...XN,X1
X3,.........X2
.
.
.
XN
```

.

I-4.
```
DEFINE MATRIX LIST
       LGTH = Ø
   IRP [,,LIST]
      LGTH = LGTH+1
   ENDIRP
   IRP [XØ,LIST2,LIST]
      COUNT = 1
      XØ
      IRP [XN,,LIST2]
          COUNT = COUNT+1
          XN
      ENDIRP
   IRP [XØ2,,LIST]
          REPEAT 1IF VZ COUNT-LGTH,STOP
          XØ2
   ENDIRP
ENDIRP
TERMINATE
```

The first IRP gets the length of the list.
The second gets the next (initially first) member.
The third processes the remainder of the list.
The fourth goes back to the beginning of the list and takes each element until COUNT = length of text.

D. OPERATION OF THE MIDAS ASSEMBLY SYSTEM

1. Preparation of a Source-Language Program

The programmer prepares his source-language program on-line via Teletype terminal, using the symbolic editing program, Editor, to type in and edit text and to store the program in English-file format on a programmer's quarter-tracks.* The file is accessible to Midas by name and version number.

2. Performing an Assembly

a. Initial Procedure

English files to be processed must be on the drum for access by Midas. Editor and Handle are used to place files on the drum; Editor will enter files from the on-line Teletype or from paper tape; Handle, from magnetic tape. Midas runs under DDT control and is called in the following way. First, the user types C"F", which puts the Call program under DDT control. Call requests the file name (Midas) then asks whether or not to start the program. If "Y" is the response to this question, Call brings Midas into core and starts it running. If "N" is the response, Call simply brings Midas into core, and to start it, the programmer must type 1Ø1"G"; to continue it, 1ØØ"G".

---

*The programmer's quarter-tracks are described in Section IV-

The condition of Midas when it is first brought into core is as follows. The current location counter is set to 11Ø and the radix indicator to 8. The macro table is empty. The symbol table contains all pseudo-instructions and a minimal list of PDP-1 instructions (these are listed in Appendix B). Additional instructions are available in an English file entitled, "System Midas Initial Symbols." Also available is a p-tape that contains a complete set of map macros and system IOT's. It is usually more convenient, however, for a programmer to construct his own p-tape of symbols that he is likely to use.

b. The Control Language

Commands to Midas are specified by control characters, such as 1 for do Pass 1 and C for continue the present pass. Some of the commands require arguments, such as the name of the file to be processed. Arguments, when required, are typed after the control character. Midas performs a requested function after line feed terminates the command string. Before the line feed is entered, an input string may be deleted by typing any 12.-bit character other than a carriage return. Spaces and carriage returns are ignored in a control string.

The control characters and argument requirements are listed on the following pages.

Control Characters	Function	Required Arguments
1	Initiate Pass 1	Name of English fil followed by a comma and the version number.
2	Initiate Pass 2	Same as for 1.
C	Continue present pass on additional file.	Name and version nu ber of additional
I	Initialize symbol and macro tables.	None
E	The argument of E represents a bit setting. Certain bit settings inform Midas to perform a special function during assembly. The bit settings and their associated functions are listed below. 16--print all characters processed 15--print an error comment on Pass 1 if the location goes indefinite 14--define undefined symbols as Ø on Pass 2	An octal number
J	Add a jump block to assembled binary program. Selects address following last START encountered	None
H	Halt Midas	None

Control Character	Function	Required Arguments
B	Set up an indexed file of binary output in programmer's storage area.	Name and version number as required for file access.
S	Set up indexed file of symbol table and macro table.	Name and version number as required by general filing.
T	Load symbol and macro table into Midas.	Name and version number under which the table is filed.

A simple assembly of a symbolic program consisting of two files would be accomplished by the following sequence of commands.

```
1  PROGRAMX,1
C  PROGRAMX,2
2  PROGRAMX,1
C  PROGRAMX,2
J
```

3. Order of Operations

The commands 1, 2, C, and J represent functions that must be performed in a certain order. The other commands represent functions that the programmer may select at various times during an assembly. These commands are discussed below with regard to the order in which they can be useful.

S--Set up File of Symbol Table and Macro Table

The "System Midas Initial Symbols" file and available p-tapes are processed on Pass 1 as individual files in a multiple file

assembly. Before continuing Pass 1 on his own program, the programmer may use <S>. Having set up these files in symbol-table format, the programmer need not perform Pass 2 on them. The file produced by an <S> command may be retrieved by command <T>.

<S> is generally also used at the end of an assembly to produce a file containing all program symbols. This file is rea into DDT to permit symbolic debugging.

I--Initialize Symbol Table and Macro Table

When assembling a very long program whose symbols may exceed the symbol-table limit or when appending a file whose symbols may be defined in an earlier segment, a user may wish to file the current symbols and use <I> to initialize the symbol tabl before continuing. On Pass 2 the first table may be read in and the table reinitialized as necessary.

The faculty for initializing the symbol and macro tables als permits the programmer to assemble a new program without restarting Midas from DDT.

B--File Binary Output

<B> is normally used once during an assembly, at the end of Pass 2. However, if the programmer wants his binary output two or more sections, he may use <B> to save a partial binar file and thus initialize the binary output buffer. Note tha while filing and initializing are separate functions with re gard to the symbol table, they are one with regard to binary output.

H--Halt

When an assembly is completed and the binary output filed, the user types <H> to halt Midas. Hitting the Break key returns control to DDT at any time.

E. BINARY OUTPUT FORMAT

All blocks, with the exception of the single-word jump block, begin with two words that indicate position and length and end with a checksum word. The maximum block length is $1\emptyset3_8$ words. The number of data words in a block is derived by subtracting the first word from the second. The checksum word contains the sum modulo $(2^{18.}-1)$ of all other words in the block, including the first two.

The first two bits of the first and second word indicate th type of block as follows:

Bits	Type
ØØ	Absolute
Ø1	Relocatable
1Ø	Library

The present manual will describe only Absolute Blocks.

The first word contains, in addition to the type-indicating bits, the address in core where the first data word is to b stored; the second, the address following storage of the la word in a block.

The pseudo-instruction WORD may be used to fabricate specia formats or to insert jump blocks without stopping the assem When Midas encounters a WORD pseudo-instruction, it termina the current block with a checksum. The arguments of WORD a appended directly to the binary output.

F. ERROR CHECKING

If Midas encounters an error in source-language coding, the assembly is interrupted and a descriptive error message printed. Depending on the severity of the error, assembly may or may not continue. The format of an error message is exemplified as follows:

(1)	(2)	(3)	(4)	(5)
USW	1ØØØ	ALPHA+1	REPEAT	GAMMA

Column (1) contains a descriptive error code; (2), the octal address at which the error occurred; (3), the symbolic address stated in terms of the last address tag seen. Column (4) contains the last pseudo-instruction symbol or macro name Midas encountered. Column (5), used only in errors involving symbol definition, contains the offending symbol.

The error codes and the conditions with which they are associated are listed on the following pages, indicating action on or impossibility of continuation of the assembly.

Error Designation	Condition Causing Error	Action Continuat
Undefined symbols		
USα	Undefined symbol (α indicates where found):	All unde: symbols evaluate zero.
C	in a constant	
D	in size of dimension array	
F	in OFFSET count	
I	in argument of ØIF or 1IF	
L	in a location assignment	
M	in storage word generated by a macro call	
O	in argument of EQUALS or OPSYN	
P	in a parameter assignment	
R	in the count of a REPEAT	
S	in the argument of a START	
T	in a multi-syllabic address tag	
W	in a storage word	
UWD	undefined symbol in argument of a pseudo-instruction	
Relocation Errors		
IRα	Illegal relocation; α identifies where the error was found with designations as listed for undefined symbol	Reloca

Error Designation	Condition Causing Error	Action on Continuation
Multiple Definitions		
MDT	Multiply defined tag	Original definition retained
MDV	Multiply defined variable (a symbol previously defined as other than a variable appears with a #)	Original definition retained
MDD	Multiply defined dimension (a previously defined symbol used as an array name)	Original definition retained
Other Errors		
MND	Macro name disagrees (the argument of a terminate disagrees with the name being defined)	First name used
ICH	Illegal character	The character is ignored
ILF	Illegal format	Characters are ignored until the next tab or carriage return
IPA	Improper parameter assignment. (The expression to the right of the equals sign is inadmissible.)	The assignment is ignored.
VLD	Variables location disagrees. (The pseudo-instruction VARIABLES has appeared on Pass 2 at a different location than on Pass 1.)	Condition ignored

Error Designation	Condition Causing Error	Action o Continuati
Other Errors cont.		
LGI	Location gone indefinite	If the app priate bit set (by Mi control <E LGI is pri on Pass 1.
IRX	Illegal operation with relocatable symbols (e.g., logical operations)	
PNT	Not an error. Result of PRINT pseudo-instruction	

In the event of the following error conditions, assembly can continue.

CLD	Constants location disagrees. The pseudo-instruction CONSTANTS has appeared on Pass 2 in a location different from Pass 1. All constants syllables have been assigned incorrect values
TMC	Too many constants (The pseudo-instruction CONSTANTS has been used too many times in one program)
TMP	Too many parameters (The storage reserved for macro-instruction arguments has been exceeded)

Error Designation	Condition Causing Error
TMV	Too many variables (The pseudo-instruction VARIABLES has been used more than 1Ø times in one program)
SCE	Storage capacity exceeded (symbol table and macro table full, too many constant words used)
IAE	An error which cannot be diagnosed--often due to improper operation of Midas

II. EDITOR

Editor is an on-line text-editing program for preparing sou language files* to be assembled by Midas. A set of control commands permits the user to select from a variety of Edito functions. Text entered, using Editor, is stored in a buf on the drum and may be changed before it is filed. Any te occurring between carriage returns is referred to as a lin Editor's text buffer. Lines may be addressed numerically symbolically. Legal Midas symbols that appear at the begi of a line followed by comma, equals sign, or colon are sto by Editor in a limited symbol table. Lines of text may be dressed relative to any of these symbols. For example, if programmer had appended the following text to the buffer,

```
        LAC ABC
        SAS DEF
ADDR1,  JMP GHI
ADDR2,  JMP JKL
```

he could refer to the first line as ADDR1-2 or ADDR2-3.

Characters entered via Teletype are stored in Internal Cod Editor can also convert to Internal Code a symbolic paper tape prepared on a flexowriter in Friden code.

---

*Files produced by Editor are referred to as English files

A. CONTROL CHARACTERS

"A" When the user types "A", all subsequent input is appended to the text buffer until he types "altmode". If "A" is preceded by a location, the test is appended after that location; otherwise, it is appended at the end of any text that has been entered.

"C" "C" changes a line, "C", preceded by a location and followed by the new text for that line, will exchange the new text for the old. Several lines may be changed at once if "C" is preceded by two locations separated by a semicolon, as illustrated below:

LOC1;LOC2"C"

"K" "K" is used to delete a line. One or two locations may precede "K", as with "C". Line numbers are updated.

"P" "P" commands Editor to print lines. "P" may also be preceded by one or two locations for printing specified lines. If no location is given, the entire text buffer is printed.

1"H" Halt Editor

B. PAPER TAPE COMMANDS

(All commands that read paper tape check parity and type "ILI if illegal parity is found. Internal Code 12 (*) is substituted for the illegal character and reading continues.)

"R"	"R" appends an Internal Code tape from th reader, clearing the reader buffer. "R" be preceded by a location that, as with " indicates where text should be appended.
"Y"	"Y" appends a concise tape from the reade clearing the reader buffer.
"T"	"T" punches a paper tape in Internal Code and takes arguments as for printing.
N"Z"	N"Z" punches N lines of tape feed.
"W"	"W" verifies a paper tape against the buf If the tape was punched with limits, the limits should be supplied for "W". In ca of failure the next 100_8 characters follc the error are printed.

C. FILING OPERATIONS

S"F"	Save the current file.
U"F"	Unsave (append) a file. Name and version number of the file must be given using "N" and "V" as described below.
SYMBOL "N"	This command names a file to be saved or unsaved. If the file is saved, the symbol preceding "N" is entered as the file name in the programmer's index.
N"V"	"V" specifies the version number, N, of a file.

D. SPECIAL CHARACTERS

@ — @ represents the last line of text in the Edit buffer. For example, if the end of the text i

```
        LAC ABC
        JMP BEGIN
START BEGIN
```

typing

```
@-1"A"
CONSTANTS
VARIABLES
⊕
```

changes the text to

```
        LAC ABC
        JMP BEGIN
CONSTANTS
VARIABLES
START BEGIN
```

LOC/ — / causes the location preceding it to be type and "opened." When a line is opened, any cha acter typed except rubout, @, line feed, or carriage return signifies the beginning of a change. Everything typed until an altmode is inserted in place of the open line. If, usin backslash or rubout, the programmer reduces t change to nothing, the line remains unchanged

RUBOUT — When appending text, rubout kills the current line. When entering a command, rubout kills the current argument.

**** — When appending text, \ deletes the last chara

+,-,space — Arithmetic operators

. — The most recent line referenced may be represented by <.>. Thus, if one had printed line 1Ø and wanted to change it, one could type .'

<.> is also used, as in DDT and Midas, to denote a decimal number.

EXPR=	If the user types EXPR=, Editor gives the number of lines (in octal) from the beginning of buffer to line "EXPR."
↑	The line preceding that last referenced may be addressed by ↑. ↑ is equivalent to <.-1>.
carriage return	Carriage return is equivalent to <.+1>, opening the line following the last line referenced.

Note that expressions may not have more than one symbol (@ and . count as symbols). Thus, A+1Ø-3+6-11-12 is acceptable, while A-B is not.

III. DDT

The on-line symbolic debugging program, DDT, used in the Hos pital System, is an adaptation of the DDT program originall developed at the Massachusetts Institute of Technology for non-time-shared-use on the TX-Ø computer. The usefulness of on-line debugging program may best be illustrated by a desc tion of the labor involved in debugging without one. Let us assume that the programmer has a binary program, a symbolic listing, a storage map, error comments from the program run, and a memory dump. Knowing where the program stopped, the p grammer will trace the program back from that point. When h has located an error, he must correct his symbolic program, reassemble it, and run the new binary. This process must be repeated until all errors have been remedied.

On-line, however, using DDT, the programmer may examine and modify registers, try out modifications, pre-set flow inter· ruptions at suspected trouble spots, and examine and modify registers before continuing the run. When patches or change are successful, the current binary configuration may be sto for further debugging.

DDT offers a wide range of debugging aids. Memory searches for a particular word or for words affecting a particular l tion may be made within selected limits or over an entire c Various input and output modes may be selected. The DDT sy table permits the programmer to debug using the symbols fro his original source program. Initially, the DDT symbol tab contains only standard machine-instruction mnemonics. A Mi symbol table may be added to this, as may individual symbol defined using DDT symbol definition functions. DDT perform

functions in response to a set of control characters. Special registers in DDT are also available to the user for setting various parameters.

DDT makes extensive use of the facilities of the Executive program. A user program to be run under control of DDT is written in binary form on the Fastrand drum in 82 fifty-word blocks. DDT, when it is running, can read into core any one of these blocks and modify the contents thereof. DDT also has access to symbol tables stored on the drum. Thus, DDT can translate the binary information that it finds in symbolic form and use symbols to calculate the binary value of a symbolic address. When a user program runs, the binary coding stored on the drum is read into the core originally occupied by DDT, and the DDT is written out on the drum. Control is then transferred to the user program at the program counter value contained in DDT. When control is transferred back to the DDT program, the user program is written out in 82 blocks, and the DDT image is read in. This transfer occurs whenever the user program tries to execute an illegal operation code or when the running user program is interrupted from the keyboard.

DDT occupies an entire core, referred to as core $\emptyset$. The user may have on the drum up to three user-program core images under control of a single DDT. When the user types "G", "P", or "X",* DDT commands that initiate program execution, DDT is written out on the drum. Then the desired core image is read into core and started. The user's cores are referred to as 1, 2, and 3. When the user program is running, six registers in lower core are reserved for communication with DDT.

---

*Quotation marks imply that the Control key is held down while the letter is typed.

A. REGISTER EXAMINATION

The operations described in this section permit the program to examine and modify registers in his program and the spec registers in DDT, listed in Section III-B. These registers may be addressed symbolically or numerically. A current lo tion counter is maintained by DDT for use in sequential reg examination. This counter contains the address of the regi to which DDT is currently pointing.

With regard to register examination we discuss the opening closing of registers. When a register is opened, its conte are printed out, and the register is available for modifica Further access to the same register requires another openir operation. A register is closed when a terminator is typed

Each time DDT types out information or accepts typed in inf mation, the current quantity register, "Q", is set equal to 18.-bit value of that information.

In symbolic addressing mode each address is represented as expression starting with the value of the nearest precedin symbol plus an integer. Note that only positive symbolic a dresses are used for output, although negative symbolic va may be used for input. Similarly, when the DDT program pr out the quantity contained in a register, it expresses tha quantity as an operation code and an address and may thus out the contents as an expression containing two symbols.

/ Slash is the basic register examination cha acter. An expression followed by / specifi the 12.-bit memory address which is to be

"opened." The current location is set to this value. For example, if the user types

MEMAD+3/

DDT will type a tab (three spaces), print the contents of the register three locations passed MEMAD, and type another tab. The mode in which the contents are printed is specified by the contents of the mode register and the radix register. The user then has the option of modifying the contents by simply typing in another expression or of closing the register unchanged.

If the user types a slash with no preceding expression, the register specified by the low-order 12. bits of "Q" is opened. The current location counter is not changed. Thus,

MEMAD/ LAC ABC /

opens ABC. If the next register in sequence is called, MEMAD+1 (not ABC+1) is opened. This facility is important for following indirect chains.

: Colon has the effect of / for 16.-bit addresses. The number of the core image referred to by the expression preceding colon is stored in register C#, one of DDT's special registers.

line feed When the user types line feed, DDT "closes" a register, inserting any modifications previously made on that line.

[(shift K) [following an expression opens the register and specifies that the contents be typed out in numerical form, regardless of the current mode setting.

] (shift M)] opens the specified register and causes the contents to be typed out symbolically regardless of the mode setting.

carriage return A carriage return serves to close a register and to open the next in sequence.

↑ ↑ closes a register and opens the preceding

ENTER Depressing the ENTER key commands DDT to cl the opened register and to open the registe specified by the lower order 12. bits of "Q resetting the current location counter to t value.

For example, the sequence

```
1ØØ/   LAC ABC
```

followed by ENTER, opens register ABC and c the sequence so the next register accessed carriage return is ABC+1 rather than 1Ø1.

' Single quote following an expression will o a register for modification without typing contents.

> > closes a register and opens the register specified by the low order 12. bits of "Q". > is similar to a slash with no argument, i that both leave the current location unchan Using >, however, the programmer may also modify the currently open registers.

B. SPECIAL REGISTERS IN DDT

Certain registers in DDT itself are accessible to the user for examination and modification. Their contents and their use within DDT are described below:

M# — M# contains the mask used in word searches. M# initially contains 777777. Search operations, discussed later, examine only those bits in user-program words that correspond to a one in M#. Search limits are continaed in the two registers following.

M#+1 — M#+1 contains the lower limit of the search. Its initial contents are 1ØØ.

M#+2 — M#+2 contains the upper limit of the search and is set initially to 7777. These two limits are also used in connection with the loading functions, "Y" and "V", discussed later.

T# — T# contains a pointer to symbol-table information kept by DDT. This information consists of sets of four registers each, created when a symbol table is read in. The first two registers contain the lower limit and upper limit, respectively, of the core range served by a symbol table. The last two of the set contain drum pointers to the table.

F# — Registers relative to F# are used by the "F" operations in DDT, which reference files. The information contained in these registers is as follows:

F#-2 mask for F"F" (find a file)
-1 file type
+Ø left half of file name
+1 right half of file name
+2 version number
+3 time and date
+4 spare
+5 location of file (pointer to toc block)
+6 location of macro table in the case of symbol files.

The use of these registers is described later with reference to the filing operations.

C# — C# contains the core image setting in the top 6 bits. The other 12. bits contain the address at which control returned to DDT. C# is change whenever the control character <:> is used to specify a 16.-bit address or when "C" is used t select a core image.

B# — B# contains the first breakpoint location. Fou breakpoints may be set at one time. B# throug B#+3 contain their locations.

G# — G# contains the number deposited by DDT in eac register of a core image by the control operation "Z" or when a new core is referenced.

E# — E# contains specifications for floating point output. The top half of the word contains the number of significant digits, and the bottom half contains the number of digits typed after the decimal point. The specified number of digits will be typed no matter what accuracy i requested. E# is initially set to 1ØØØ7 (eigh significant digits, seven digits after the dec mal point).

C. DDT OUTPUT CONTROL

Information may be entered by the user or typed out by DDT in several different forms. A quantity or a memory address may be represented by an arithmetic expression containing a symbolic or a numerical value. While these forms may be used interchangeably by the user, DDT output will be printed according to the mode setting and the radix value. When DDT is first read in, the mode is set for symbolic addressing and symbolic quantity printout. The control characters discussed below are used to change the setting.

A!	Mode is set for printing locations in absolute numerical form.
R!	Mode is set for printing locations in symbolic form.
C!	Mode is set for printing the contents of registers as numerical values.
F!	Mode is set for typing contents as floating point numbers. The addressed register plus the next register in sequence are treated as a single entity. Thus, carriage return in this mode opens the location plus two rather than one, and ↑ the location minus two.
I!	Mode is set for interpreting contents as Internal Code and printing alphanumeric representations.
S!	Mode is set for printing the quantity contained in a register symbolically (as an operation code and a symbolic address).
"R"	Resets radix for numerical output as follows:

2"R"--binary output
12"R"--decimal output
1Ø"R"--octal output

Ø"R" and 1"R" are illegal, but any other valu
up to 12 is permitted.

The above characters serve as "permanent" mode switches; the output mode may also be switched for a single register print The following characters will cause DDT to type "Q", the las quantity typed by DDT or the user, in the mode specified.

=	"Q" will be printed as an integer of radix R.
←	"Q" will be printed symbolically.
?	"Q" will be interpreted as Internal Code and printed as alphanumeric characters from left right with spaces deleted.
"S"	"Q" will be interpreted as squoze code and printed alphanumerically.
$	The contents of the current location plus th following location are printed as a floating point number, with significance and accuracy specified by E#+1.
"O"	"Q" will be printed as a time and date.

D. SPECIAL INPUT SYMBOLS

The characters below inform DDT to interpret typed input in a specified mode.

"	A double quotation mark following three letters or numerals causes their Internal Code values to be entered.
.	A period immediately following an integer specifies it as a decimal number.

The two symbols below have values and may be used as part of an expression.

. (period)	The value of the current location counter (12. bits).
"Q"	The value of the last quantity typed by DDT or by the user (18. bits).

The following input symbols are combining operators and are used to form expressions for evaluation:

Space or +	arithmetic plus
-	arithmetic minus
"U"	Boolean inclusive OR on a bit-by-bit basis
&	Boolean AND on a bit-by-bit basis

E. SYMBOL DEFINITION

The following control characters are used to define symbols to specify the range of applicability of a symbol definitior A global symbol is one that is defined for all core images u having the status of the DDT initial symbols. Local symbols are defined only for the core image being addressed at the t of their definition.

,	A comma immediately following a symbol defin that symbol locally as the 12.-bit current location.
>	The preceding symbol, SYM, is defined locall as the low order 12. bits of "Q". For examp the sequence

```
ABC/   LAC 1ØØ   DEF>
```

assigns the symbolic address DEF to location

(SYM)	The parenthesized symbol is defined globally the value of "Q". Thus EXP (SYM) sets the s to the value of EXP.

F. SEARCHING OPERATIONS

These operations are used to determine where, in a given section of memory, a word or part of a word is present. DDT uses the mask in M# and the limits in M#+1 and M#+2 to define which part of the word and which section of memory are to be searched. Since only bits corresponding to ones in the mask are considered in the search, it is possible to search for occurrences of a portion of a word as, for example, an instruction, a negative sign, etc.

EXPR"W" DDT will search memory for registers whose contents have the value of EXPR masked by the contents of (M#). The location and contents of every such register are printed out. Use of "W" without an argument is an error.

EXPR"N" "N" works in the same way as "W" to locate all registers that do not contain the value of EXPR. For example, the command Ø"N" will cause all registers that do not contain Ø to be printed.

EXPR"E" DDT searches memory for registers that contain an effective address equal to EXPR. Indirect addressing chains are followed to a depth of 1ØØ. The search ignores IOT instructions as well as instructions in the SHIFT, SKIP, and OPERATE groups. If a JDA EXPR exists, an "E" search for EXPR+1 will locate it.

G. FILING OPERATIONS

DDT filing operations use information contained in the specia registers described in relation to F#.

S"F" The save-file command is used to write core im ages onto the drum. DDT will write the core a which it is looking (as specified in register on a programmer's quarter-track, type the file core image's drum address, and store the addre in register F#+5.

A"F" The add-file command adds an item to the programmer's index, using information contained in registers F#-1 through F#+6.

U"F" U"F" "unsaves" the core image whose drum location is in F#+5, bringing it into the core specified in C#.

M"F" When a program that is not running under DDT halts on an illegal instruction (indicating a program error), the core image is written on the message tracks. The name and location of the core image on the tracks are printed on t Soroban typewriter. With this location enter in F#+5, M"F" will unsave the core image from the message track.

F"F" F"F" is used to find the drum address of the file specified in registers F#-1 through F#+2 The address is typed out and also stored in F#+5. The mask in F#-2 is used in this searc The file found is one whose index entry match the contents of those words in F#-1 through F#+6 for which the bit in the mask is set. T mask is initially 17, for the usual case of name (two words) and version number (one word To ignore version number, set the mask to 7. If a file is found, its index entry is moved F#-1 through F#+6.

P"F" Proceed with search. If there is more than one file with a particular name and number, the command F"F" will find the first appearing in the index. P"F" is used to continue the search in order to locate the desired occurrence.

D"F" Delete a file. The file whose type, name, and version number are specified in F#-1 through F#+2 is found, and its type is changed to -Ø in the programmer's index.

L"F" Programs are called from the library by L"F". The program whose library address is in F#+5 is read into the current core image.

C"F" This command is also used to call library programs. When the programmer types C"F", DDT types C? and waits for the user to type in the program name. Input is terminated by ENTER. DDT then types <S?>. If the user wishes DDT to start running the called program, he types Y; otherwise, he types N.

H. LOADING THE PROGRAM

In order to facilitate debugging segmented programs, up to three core images may be under DDT control at one time. The current core is changed by <:> or "C". Core images are not generated until they are needed. At the first reference to a core, the core image is generated on the Fastrand drum, and with the exception of some registers in lower core (such as Teletype number, programmer's index pointer, etc.), the core image is filled with the contents of G#. "Z" may be used at any time to fill a core with the contents of G#. If no arguments precede "Z", G# is loaded from register 1ØØ to register 7777. G# contains Ø when DDT is first called but may be char to some other quantity. "Z" preceded by two numbers separate by a semicolon (as, 1;2) will deposit the contents of G# with limits, from 1 to 2 in this case.

The commands below are used to load a program prior to debugging with DDT.

"Y" — "Y" will read in a binary file or a portion of a fi from the drum (using the limits in M#+1 and M#+2). The name and version number of the desired file mus be in registers F#, F#+1, and F#+2.

1"Y" — "Y" preceded by 1 reads a paper tape, also using M# and M#+2.

"T" — "T" reads a symbol-table file, the name and version number of which is in F#, F#+1, and F#+2. Before r ing symbols, the user must specify core locality by typing the lower limit followed by colon and the up limit.

For example,

1ØØØØ:2ØØØØ"T"

will read a symbol table local to core 1. These symbols will be undefined for other locations.

"V" "V" is used to verify a core image against a binary file already located in core. The rules for "Y" hold.

1"V" Preceded by 1, "V" verifies a core image against a paper tape. The comments for 1"Y" apply.

I. RUNNING THE PROGRAM

The operations in this section allow the user to control the running of his program, starting it at any point and interrupting it at will. Note that DDT does not run the program interpretively but functions as a supra-executive, trapping a whole set of commands.

"G" When control character "G" preceded by a location is typed, DDT writes itself out on the programmer's tracks, brings the user's program into core memory, and starts it running at the location specified. "G" is illegal in DDT's core.

"B" As many as four breakpoints may be pre-set in a sing program run. Registers B# through B#+3 contain the locations. To insert a selected breakpoint at a pa ticular address, type the number of the breakpoint (Ø, 1, 2, or 3), followed by semicolon, followed by the address in the program at which the breakpoint to be placed, followed by "B". Zero operation code are deposited in these locations, and the original operation code is stored in the pertinent B# regist When any breakpoint location is reached during program execution, control is transferred to the Execu tive as it is for all illegal instructions. Execut then transfers control back to DDT. DDT replaces t correct instruction at that breakpoint location and types out the location of the breakpoint and its co tents in the form:

1ØØ) LAC A

To remove a selected breakpoint, type the number of the breakpoint, followed by semicolon, followed by Typing "B" alone removes all breakpoints.

"P" "P" proceeds with execution of a program after a breakpoint interruption. Zero operation codes are again placed in all breakpoint locations. The correct instruction at the proceed location is saved in register 1Ø5. The next two registers contain jump instructions to the location +1 and +2. Thus it is possible to put a breakpoint at a register that proceeds at either relative location. If procedure after an interrupt begins more than two registers away, it cannot be performed. Note that breakpoints may be placed at locations whose contents are modified in their address part but not in their instruction parts. "P" may be preceded by an expression specifying the number of times to run through the breakpoint before interrupting. 1Ø"P", for example, will run through the breakpoint ten times.

"X" DDT can execute a single instruction. The instruction to be executed must be typed immediately before "X". LAW 1ØØ"X" will put the number 1ØØ in the accumulator for example. "Q""X" may be used to execute the last quantity typed. Control remains with DDT unless the instruction is a jump to the user's program.

J. PUNCHING OPERATIONS

The following operations permit the user to have output pu
on paper tape.

-Ø"L"	This command punches a loader and sets the mode to "non-zero dump," i.e., registers wh contents match the contents of G# will not punched.
1"L"	1"L" causes DDT to punch a loader.
MIN;MAX%	The character % causes DDT to punch checksu data blocks. MIN and MAX specify the limit the registers to be punched.
*	* will punch the contents of an opened regi as a checksum block.
EXPR@	The control character @ punches a jump bloc the end of a paper tape. The expression pr ceding it specifies the starting address of program.

K. ERROR COMMENTS

In response to most input errors, DDT types a question mark. If a typed-in symbol is not defined in the DDT symbol table, for that core region, DDT types U. If question mark is typed because of an illegal control character, only the illegal character is ignored. If it is because of an illegal operation (such as "G" in DDT core) all input is ignored. In the case of an undefined symbol, only that symbol is ignored.

ABN followed by two numbers indicates a Fastrand error. The first of the two numbers is the register in DDT at which the error occurred; the second indicates the type of error.

SUM is printed if a checksum error occurs during execution of "Y" or "V".

L. MISCELLANEOUS FUNCTIONS

Rubout	In case of a typing error, rubout may be used to delete all input preceding it.
"D"	"D" will turn a Teletype off temporarily.
1"H"	This command halts DDT and releases the drum field for other use. The file and index are of course lost.

IV. HANDLE

Midas, Editor, and DDT all use the drum for temporary storage. This stored information is lost when DDT is halted. Any of the information that will be needed in the future must be saved on paper or magnetic tape. DDT can produce paper tape; use of paper tape, however, is often inconvenient. The Handle program is designed to retrieve files from the drum and store them on magnetic tape. When needed again, these files may be retrieved from tape and placed on the drum for use at the start of the next programming session. Handle, like Editor and Midas, is called under DDT, using the C"F" feature explained in the previous section. A discussion of the file system follows.

A. PROGRAMMING SYSTEM FILES

Since all of the Programming System's programs are run under control of DDT, the system file structure is superimposed on the drum storage space allocated to DDT. When a user calls DDT, the Executive Program assigns to him a "Programmers' Quarter-Track." This quarter-track is a scratch area on the drum that is held inviolable for that programmer until his I program halts. The capacity of this scratch area is approxi mately one million bits, or some 12-1/2 core-loads of memory

Should the programmer require more space than is allowed on single quarter-track, the Executive will automatically assig additional ones as needed, up to the limit of the storage space allocated on the drum for this purpose.

The output produced by each of the Programming System's pro grams must be referenced by the others. For example, a sou language program (an English file) produced with Editor mus be available to Midas for assembly. Any output which must addressed by some other program is written as a file on the programmer's quarter-tracks and is assigned a name and a ve sion number. So that files may be referred to by name and number, the Programming System maintains an index of the fi associated with each running DDT. This index, called the Programmer's Index, contains an eight-word entry for each f Figure 2 illustrates the format of an entry in the Programm Index. Each such index is available to all programs in the Programming System. All indexed files are available to Ha

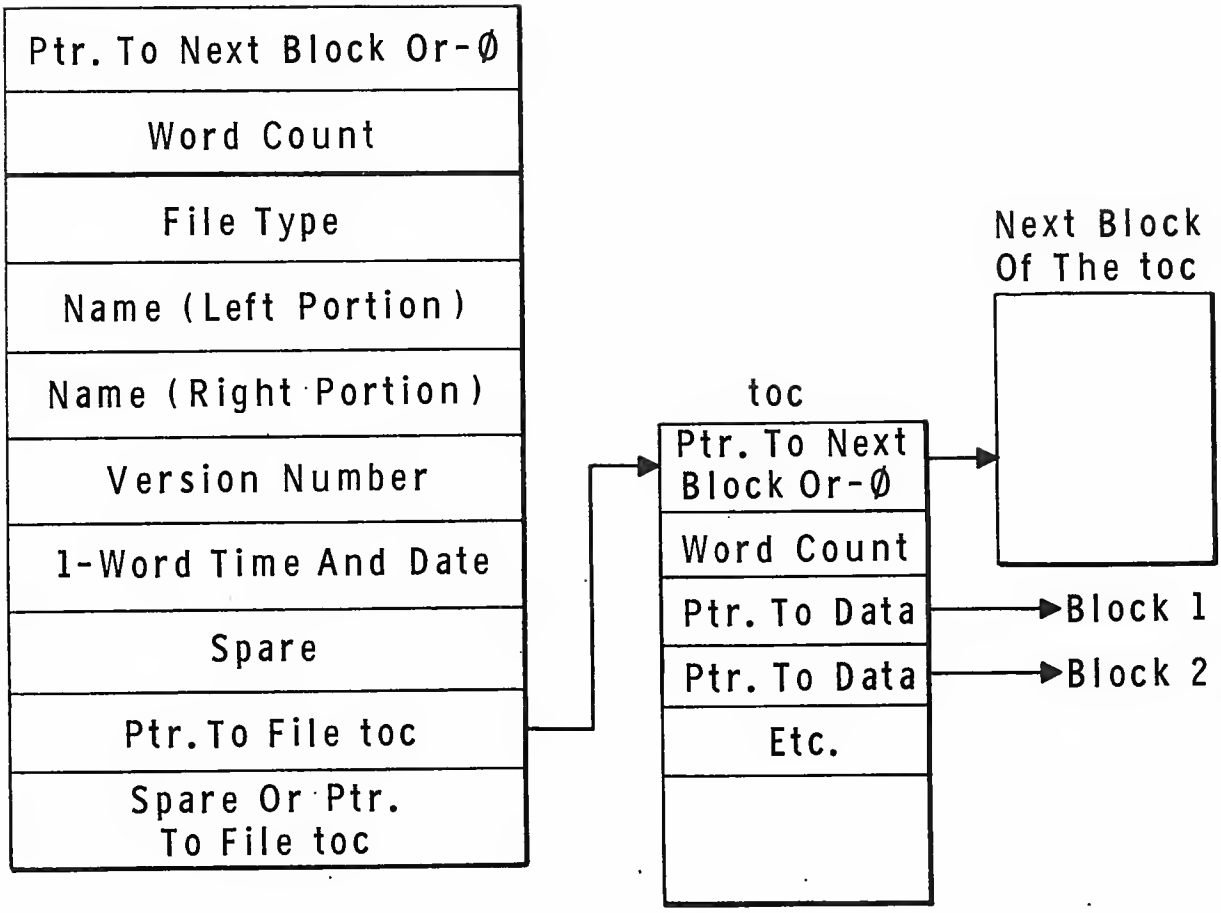


FIG. 2 THE PROGRAMMER'S INDEX

File types are as follows:

Type Ø--Binary files (paper tape images), produced by Midas, read by DDT.

Type 2--Core Image files (binary contents of a 4096-word core produced and read by DDT.

Type 3--English files, produced by Editor, read by Midas.

Type 4--Symbol Table files, produced by Midas, read by Midas and DDT.

English, binary and core-image files are written as "toc'd items." The toc (table of contents) is a drum item that contains a list of pointers to data blocks. The address entry i the Programmer's Index is a pointer to the toc for a given file.

In addition to the list of pointers in a toc, there is at least one extra word (overhead) containing the length of the toc. Core images are standard rewriteable items with four words of overhead.

Symbol tables are filed in the format in which they are used Midas during an assembly. Two addresses are entered for a symbol file in the Programmer's Index; one address is a poin to the symbol table, the other a pointer to the macro table. The format for symbol files is illustrated by Figure 3.

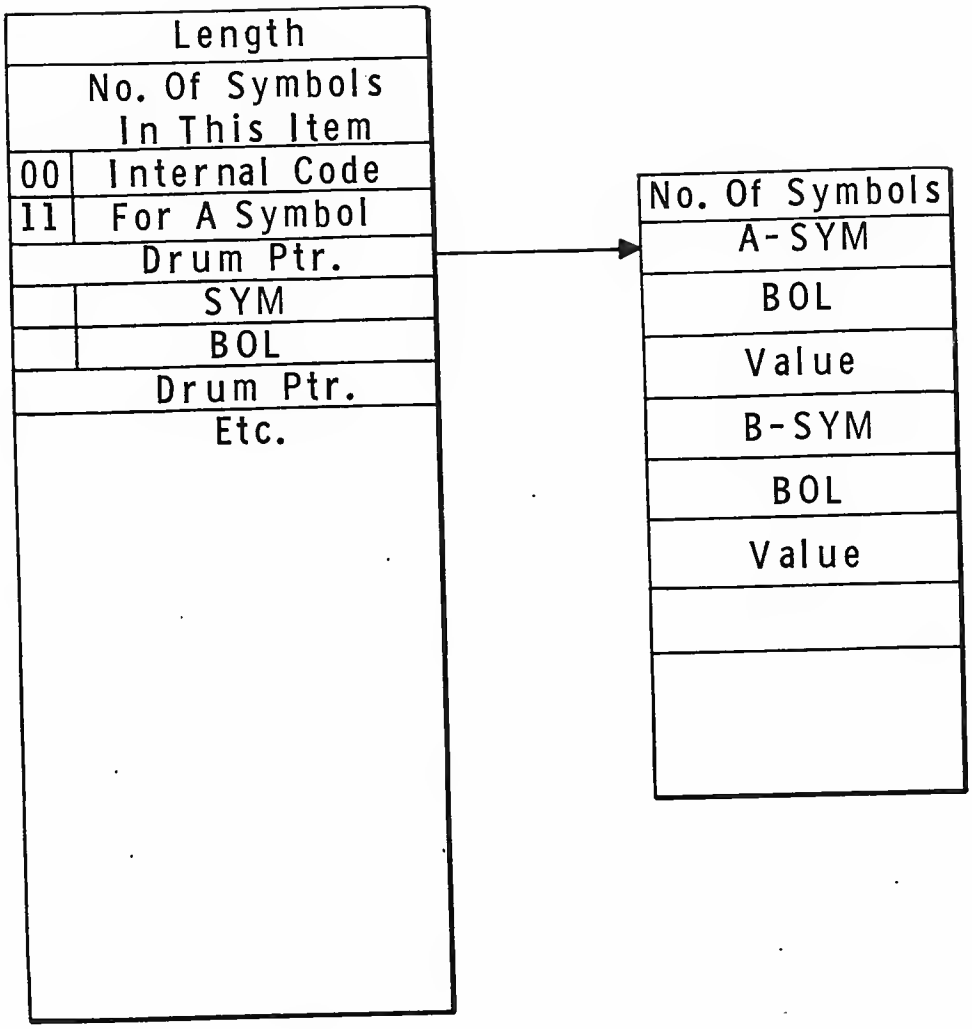


FIG. 3 SYMBOL TABLE FILES

B. HANDLE COMMANDS

After a typical programming session, it is most likely that the programmer will have a set of programmer's quarter-trac littered with various source-language versions of the same program, remnants of abortive assemblies, and obsolete as we as active symbol and macro tables. In addition to his own files, he may be using files that he has "borrowed" from ot programmers using the system. Those things he wishes to sa he will organize and index on magnetic tape; those he wishe to discard, he will "kill" or allow the system to discard. Things he has borrowed from others he must copy into his ow files if he wishes to be assured of their permanence. The Handle program contains a set of commands that the programm uses to implement this housekeeping task.

The commands to Handle fall into two categories:

1) file manipulation commands
2) magnetic tape manipulation commands

Using the first category, file manipulation, the programmer add a specific file to his index or delete one from it. He have a list of file names printed, having the option as eac typed of retaining or discarding it. Other Handle commands low the programmer to find the absolute drum address of one his files or to add someone else's files to his collection.

In the second category, magnetic tape manipulation, Handle provides the programmer with a means for writing files on t and reading them from it, as well as for having file names printed from tape.

1. The Control Language for Files

The Handle control language for file manipulation consists of one-letter commands. These, with their mnemonics, appear in the following table:

Table 1

Handle Control Language for Files

Code	Mnemonic
A	ADD
B	BREAKDOWN
D	DELETE
E	EDIT
F	FIND
H	HALT
K	KILL
L	LIST

When a terminator is entered, Handle executes the command typed, ignoring any extra symbols preceding it. Thus, mnemonics of commands, such as those used in the tables or the following paragraphs, are acceptable input. In all cases, files are referred to by name, version number, and type. Th commands are described below in detail.

A ADD

ADD NAME,VERSION,TYPE⊕

Files that have been added to the program mer's quarter-tracks via the DDT program files taken from another programmer's ind are added to the user's index through the A command. Note that this command merely updates the index and does not transcribe the file from one programmer's track to a other's. In order to use a "borrowed" fi the user must actually transcribe it, sir it will desappear when the other programmer's DDT is halted. The transcription capability is provided in DDT.

D DELETE

DELETE NAME,VERSION,TYPE⊕

The DELETE command does not actually rem and information from the drum, but sets a flag associated with that NAME, VERSION, and TYPE in the programmer's index. Whe the programmer then transcribes this fil using the index, that particular NAME, VERSION, and TYPE will no longer appear nor will the data associated with it.

E EDIT

The EDIT command causes Handle to list t entire contents of the programmer's inde one file at a time. After each file nam typed, the program waits for input. If user types only ⊕, that particular file retained; if the user types K⊕, it is ex actly the same as deleting that particul NAME, VERSION, and TYPE. Out-of-date fi may thus be removed from the index.

K KILL

KILL deletes an entire programmer's index. The drum address of the programmer's index is typed out when a KILL is executed in order to make it possible for the programmer to recover in case the K was executed by accident.

L LIST

The LIST command is essentially the same as the EDIT command except that Handle, instead of pausing after each line of typeout, continues until the end of the programmer's index.

F FIND FIND NAME,VERSION,TYPE⊕

The FIND command is used to determine the drum address of a file (the two drum addresses in the case of a symbol file). The FIND command is particularly useful in communicating drum addresses to some other programmer who wishes to borrow a file. Given the associated address, he can transcribe that file to his own tracks using DDT.

B BREAKDOWN

The B command is similar to the LIST command except that the drum addresses associated with the individual files are also typed out.

H HALT

HALT causes Handle to execute the HLTGUD IOT and thus returns control to the DDT under which it has been running.

2. The Control Language for Tape Manipulation

The commands tabulated and discussed below are used for magn tape manipulation. Before using these commands, the user mu contact the computer room in order to have his tapes mounted and a tape unit assigned to him.

Table 2

Control Language for Magnetic Tape

Code	Mnemonic
O	ORIGIN
P	PRINT
R	READ
S	STOP
W	WRITE

O ORIGIN

ORIGIN signals the tape unit to rewind t the "change tape" position. This comman is used to terminate tape activity. Sin it requires operator intervention at the computer room to move the tape from the change tape to the "load point" position executing an ORIGIN is a safety measure, holding the tape locked until the operat can be notified to remove it.

S STOP

The STOP command writes three end-of-file markers on the tape and then rewinds it to the load point position. This command is used only with new tapes and provides a reference starting point for new tape files.

W WRITE

The WRITE command advances the tape until it finds three end-of-file marks (obtained with the STOP command) and then writes out all the programmer's current files. To determine which files are current, Handle reads the programmer's index and transcribes onto magnetic tape only those files that have not been deleted or killed. When the transcription is completed, Handle writes three end-of-file marks, types out the block address of the index, and rewinds the tape.

P PRINT

PRINT lists all of the file names on a magnetic tape and their associated magnetic tape block address. No transcription takes place to the drum, and the tape is rewound at the completion of the PRINT function. PRINT is useful for finding specific files prior to the READ command.

R READ

The READ command requests the user to type the block address of the index. Handle then locates the index and transfers it and the files to which it refers from tape to drum. A new programmer's index is compiled on the drum, and its address is stored in register 24 of DDT. At the completion of the READ command the tape is rewound.

It should be noted that Handle attempts to deal with tape or drum errors. If an error occurs which Handle cannot manage, the code ABN is typed, and the type of error is described.

Appendix A. Midas Character Set

I. Alphabetic

Letters (A-Z)
Digits (Ø-9)

II. Punctuation

Character	Function(s)
,	a) indicates address tag mod (2 b) separates elements of a List c) terminates count of a REPEAT
:	indicates address tag mod (2^{16})
=	equates symbol to the left with expression to the right
/	a) terminates location assignme b) introduces comment c) introduces list of macro-ins arguments to be generated d) terminates a conditional
()	enclose a literal
[]	expression enclosed specified fo syllable function
#	denotes symbol as a variable
$	indicates global symbol*
tab and carriage return	a) word terminators b) varying meanings according t context.

*Relocatable programming feature.

III. Combining Operators

Product Operators

"T"	folded integer multiplication
"X"	logical disjunction (exclusive OR)
"U"	logical union (inclusive OR)
"A"	logical intersection (AND)
"Q"	quotient
"R"	remainder

Additive Operators

+ or space	Addition, mod $2^{18.}-1$
-	Addition of the one's complement

IV. Illegal

A. General

break
rubout
\
altmode
←

B. (except within a macro-instruction or an IRP)

!
"
"B"
"C"
wru
"F"
"G"
"H"
;
<
>
?
vert. tab
"K"
"N"
"O"
"P"
"S"
"V"
"W"
"Y"
"Z"

V. Ignored (except within a macro-instruction or an IRP)

'
EOT
formfeed
carriage return
↑

Appendix B. Symbols in Permanent Midas Vocabulary

1. PDP-1 Instruction Symbols

The following list contains all instruction included on the "MIDAS INITIAL SYMS FOR T.S." tape. Those included in the initial vocabulary are starred.

MIDAS INITIAL SYMS FOR T.S.

```
 1S=1
 2S=3
 3S=7
 4S=17
 5S=37
 6S=77
 7S=177
 8S=377
 9S=777
 AAI=740003
*ADD=400000
*AND=020000
 CAL=160000
 CLA=760200
 CLC=761200
 CLF=760000
 CLI=764000
 CLL=740010
 CLO=651600
 CMA=761000
 CMI=770000
 CML=740004
*DAC=240000
*DAP=260000
*DCH=140000
*DIO=320000
*DIP=300000
*DIV=560000
*DZM=340000
 HLT=760400
*  I=010000
 IAI=740002
 IDA=740400
 IDC=741000
*IDX=440000
 IFI=742000
 IIF=744000
```

```
*IOR=040000
*IOT=720000
*ISP=460000
*JDA=170000
*JMP=600000
*JSP=620000
 LAI=760040
*LAC=200000
 LAP=760300
*LAW=700000
*LCH=120000
 LFI=742050
 LIA=760020
 LIF=744100
*LIO=220000
*MUL=540000
 NOP=760000
 OPR=760000
*RAL=661000
*RAR=671000
*RCL=663000
*RCR=673000
*RIL=662000
*RIR=672000
*SAD=500000
*SAL=665000
*SAR=675000
*SAS=520000
 SCF=740040
 SCI=740100
*SCL=667000
 SCM=740200
*SCR=677000
 SFT=660000
*SIL=666000
*SIR=676000
 SKP=640000
*SMA=640400
*SNI=644000
*SPA=640200
*SPI=642000
 SPC=740000
 SPQ=650500
 STF=760010
*SUB=420000
 SWP=760060
```

```
*SZA=640100
*SZF=640000
 SZL=740020
 SZM=640500
 SZO=641000
*TAD=360000
 XAI=740001
*XCT=100000
*XOR=060000
  XX=HLT
```

2. Pseudo-Instructions

Symbol	Function
CHARACTER	inserts Internal Code for one characte
CONSTANTS	specifies storage areas for constant v
DECIMAL	classifies integers as decimal numbers
DEFINE	initiates macro-definition
DIMENSION	allocates storage area for arrays
ENDIRP	ends an indefinite repeat
EQUALS	establishes symbol equivalence
EXPUNGE	erases symbols from symbol table
IRP and IRPC	initiates indefinite repeat
NULL	no operation
OCTAL	classifies integers as octal numbers
OFFSET	assigns address tags as current locati counter and sets an expression whose v is the offset count.
OPSYN	same as EQUALS; Pass 1 only
PRINT	generates symbolic location printout a prints comment during assembly
PRINTX	prints comment during assembly
REPEAT	generates iterative source-language te

START	denotes end of source program and specifies starting address
STOP	ends expansion of IRP's, macro's, and REPEAT's
TERMINATE	ends macro-definition
TEXT	inserts Internal Code for character string
VARIABLES	reserves space for variables and arrays
WORD	appends word(s) to binary output block
ØIF	tests an expression; if true, value is zero; if false, one.
1IF	if true, value is one; if false, zero.

Appendix C. Teletype Code Conversion

("X" means control-X)

INTERNAL	ASCII	CHARACTER
ØØ	Ø4Ø	SPACE
Ø1	Ø41	!
Ø2	Ø42	"
Ø3	Ø43	#
Ø4	Ø44	$
Ø5	Ø45	%
Ø6	Ø46	&
Ø7	Ø47	'
1Ø	Ø5Ø	(
11	Ø51	)
12	Ø52	*
13	Ø53	+
14	Ø54	,
15	Ø55	-
16	Ø56	.
17	Ø57	/
2Ø	Ø6Ø	Ø
21	Ø61	1
22	Ø62	2
23	Ø63	3
24	Ø64	4
25	Ø65	5
26	Ø66	6
27	Ø67	7
3Ø	Ø7Ø	8
31	Ø71	9
32	Ø72	:
33	Ø73	;
34	Ø74	<
35	Ø75	=
36	Ø76	>
37	Ø77	?

INTERNAL	ASCII	CHARACTER
4Ø	1ØØ	@
41	1Ø1	A
42	1Ø2	B
43	1Ø3	C
44	1Ø4	D
45	1Ø5	E
46	1Ø6	F
47	1Ø7	G
5Ø	11Ø	H
51	111	I
52	112	J
53	113	K
54	114	L
55	115	M
56	116	N
57	117	O (OH)
6Ø	12Ø	P
61	121	Q
62	122	R
63	123	S
64	124	T
65	125	U
66	126	V
67	127	W
7Ø	13Ø	X
71	131	Y
72	132	Z
73	133	[
74	175,176,Ø33	EOM
75	135	]
76	Ø15-Ø12	CARRIAGE RETURN-LINE FEED
77		WARNING

INTERNAL	ASCII	CHARACTER
77ØØ	ØØØ	NULL or BREAK or "©"
77Ø1	ØØ1	"A"
77Ø2	ØØ2	"B"
77Ø3	ØØ3	"C"
77Ø4	ØØ4	EOT
77Ø5	ØØ5	"E" or WRU
77Ø6	ØØ6	"F" or RU
77Ø7	ØØ7	"G" or BELL
771Ø	Ø1Ø	"H"
7711	Ø11	TAB
7712	Ø12	LINE FEED
7713	Ø13	"K" or VT
7714	Ø14	"L" or FORM FEED
7715	Ø15	CARRIAGE RETURN (OUTPUT ON
7716	Ø16	"N"
7717	Ø17	"O"
772Ø	Ø2Ø	"P"
7721	Ø21	"Q"
7722	Ø22	"R" or TAPE
7723	Ø23	"S" or RDR OFF
7724	Ø24	"T"
7725	Ø25	"U"
7726	Ø26	"V"
7727	Ø27	"W"
773Ø	Ø3Ø	"X"
7731	Ø31	"Y"
7732	Ø32	"Z"
7733	Ø33	"["
7734	Ø34	SHIFT "L"
7735	Ø35	"]"
7736	Ø36	"↑"
7737	Ø37	"←"
774Ø-7743	UNUSED	
7744	134	BACKSLASH
7745	UNUSED	
7746	136	↑
7747	137	←
775Ø-7773	UNUSED	
7774	177	RUBOUT
7775-7777	UNUSED